Queens College, CUNY, Department of Computer Science
Object Oriented Programming in C++
CSCI 211 / 611
Summer 2018
Instructor: Dr. Sateesh Mane

June 28, 2018

Inheritance: Part II

- In this lecture we continue the study of **inheritance**.
- In this lecture we shall learn how to write constructors for derived classes.
- We shall also learn about the "Big Three" (copy, assign, destroy) for derived classes.
- We shall also study the use of **pointers and references** in connection with inheritance.
- We shall also briefly mention the concept of **slicing**.

1 Constructors, destructor, more about overriding

- How do we write constructors for a derived class? The answer is not so obvious.
- What about the Big Three (copy, assign, destroy)? (We shall postpone this question.)
- To answer these questions, we shall employ the example of the rectangle and square.
- For simplicity we only consider upright rectangles, i.e. horizontal and vertical sides.
- We declare a `Rectangle` class with the following properties.

  1. Default and non-default constructor.
  2. Destructor (for educational use with derived classes).
  3. Protected data members `h,w` for the height and width, respectively.
  4. Public accessor methods and public method "`area`" to calculate the area ($= |h * w|$).
  5. Public method "`print`" to print the (x, y) coordinates of the vertices.

- The class declaration is displayed on the next page.
- For educational use we print an output statement in the constructors and destructor.

1.1 Base class `Rectangle`

```
class Rectangle {
public:
  Rectangle()                               // default constructor
  {
    h = 1.0;
    w = 1.0;
    cout << "Rectangle: default constructor" << endl;
  }

  Rectangle(double ht, double wd)           // non-default constructor
  {
    h = ht;
    w = wd;
    cout << "Rectangle: non-default constructor" << endl;
  }

  ~Rectangle()                              // destructor
  {
    cout << "Rectangle: destructor" << endl;
  }

  double height() const { return h; }       // public methods
  double width()  const { return w; }
  double area()   const { return abs(h*w); }

  void print() const {                      // "print" method
    cout << "Rectangle::print ";
    cout << "(" << 0 << "," << 0 << "), ";
    cout << "(" << w << "," << 0 << "), ";
    cout << "(" << w << "," << h << "), ";
    cout << "(" << 0 << "," << h << ") ";
    cout << endl;
  }

protected:
  double h, w;                              // protected data
};
```

1.2 Derived class Square

- We now arrive at the main purpose of this exercise, which is the derived class Square.
- The class declaration is given below.
- We shall write the various functions step by step below.
- The class Square also has a Boolean data member "centered" which will be used later.
- The data is tagged as protected, so classes which derive from Square can access it.
- A square has an "edge length" as opposed to "height" and "width" and so the class Square has a public method "edge" which is not in the class Rectangle.
- **Derived classes can declare extra methods which are not in the base class.**

```
class Square : public Rectangle {          // inherts from Rectangle
public:
  Square();                                // default constructor
  Square(double e, bool c);                // non-default constructor
  ~Square();                               // destructor

  double edge() const;                     // extra method not in base class

  void print() const;                      // overrides method in base class

protected:
  bool centered;                           // data in derived class
};
```

1.3 Constructors of derived class

- The **default constructor** is written as follows.

```
Square::Square() : Rectangle()
{
    centered = false;
    cout << "Square:  default constructor" << endl;
}
```

- Objects of derived classes are constructed in sequence.
- The base class constructor is invoked first, then the derived class constructor.
  1. In the above example, ": **Rectangle()**" tells the compiler to invoke the default constructor of the `Rectangle` class.
  2. **It looks similar to the member initialization list.**
  3. **It *** is *** part of the member initialization list.**
     (a) We are initializing the "Rectangle" component of the "Square" object.
     (b) The default constructor of the `Rectangle` class initializes the values of h and w.
  4. The body of the default constructor of the `Square` class is straightforward.
     (a) The data member `centered` belongs to the `Square` class, so we initialize it.
     (b) We also print a line of output for educational purposes, to be used below.
- The **non-default constructor** is written in a similar way.

```
Square::Square(double e, bool c): Rectangle(e,e)
{
    centered = c;
    cout << "Square:  non-default constructor" << endl;
}
```

- This time, we call the non-default constructor of the `Rectangle` class.
  1. The height and width of a square are equal, so we call the non-default `Rectangle` constructor with arguments (e, e) to set the height and to equal values.
  2. The input argument c is used to initialize the value of the data member `centered`.
  3. We also print a line of output for educational purposes, to be used below.
- Actually, a derived class constructor can call any constructor it likes in the base class.
- However, it makes the most sense for the `Square` default constructor to call the `Rectangle` default constructor, and for the `Square` non-default constructor to call the `Rectangle` non-default constructor.

1.4 Destructor of derived class

- The **destructor** is written as follows.

```
Square::~Square()
{
  cout << "Square: destructor" << endl;
}
```

- **The destructor looks basically the same whether or not `Square` is a derived class.**
- The destructor of the derived class does not explicitly call the destructor of the base class.
- The compiler does that automatically for us.
- Unlike the constructors, there is only one destructor for a class, so there is no ambiguity "which destructor" to call.
- In the present example, the class `Square` does not dynamically allocate memory.
- As we know, if an object dynamically allocates memory, it should be released in the destructor.
- **Objects are destroyed in the opposite order to how they are constructed.**
  1. Hence when an object of type `Square` goes out of scope, the compiler invokes the destructor of `Square` first, to destroy the "`Square`" part of the object.
  2. Next the compiler invokes the destructor of the `Rectangle` base class, to destroy the implicit `Rectangle` object contained in the `Square` object.

1.5 Derived class `Square`: accessor methods

- In this example, the class `Square` does not override the accessor methods in the `Rectangle` base class.
- There is no need: there is nothing to change in the methods `height()` and `width()`.
- There is also no need to override the method "`area()`" because the area of a square is given by the same mathematical formula for a rectangle.
- *However, the concepts of "height" and "width" do not really make sense for a square.*
- A square has an edge length, rather than a height and width.
- Hence the `Square` class declares a method "`edge()`" to return the value of the edge length.

  ```
  double Square::edge() const { return height(); }
  ```

  1. Because the height and the width are equal, it is sufficient to return the height.
  2. Hence the method "`edge()`" is just a convenience, because for a square we speak of its edge length, not its height or width.

- The method "`edge()`" is not in the `Rectangle` base class.
- **Derived classes may declare extra methods which are not in the base class.**
- A derived class is not restricted to only override methods from the base class.

1.6 Derived class `Square`: override `print`

- We now arrive at something very interesting.
- The method `print()` in the `Rectangle` class prints the (x, y) coordinates of the vertices of the rectangle. The bottom left corner is fixed at the origin $(0, 0)$.
- For the class `Square`, we override `print()` to do something different.
  1. The class `Square` has a Boolean data member `centered`.
  2. If `centered==true`, we put the center of the square at the origin $(0, 0)$ and print the coordinates of the vertices with appropriate $\pm$ signs (see below).
  3. If `centered==false`, we use the `print()` method of the `Rectangle` base class. (Hence the bottom left corner is at $(0, 0)$).
- Hence this is our software design. ***How shall we implement it?***
  1. If `centered==true`, we override the base class method.
  2. If `centered==false`, we do not override.
- The code for `print()` is displayed below.

```
void Square::print() const {
    if (centered == false) {
      Rectangle::print();                    // call base class method
    }
    else {
      double d = w*0.5;
      cout << "Square::print ";
      cout << "(" << -d << "," << -d << "), ";
      cout << "(" <<  d << "," << -d << "), ";
      cout << "(" <<  d << "," <<  d << "), ";
      cout << "(" << -d << "," <<  d << ") ";
      cout << endl;
    }
}
```

- **We call the base class method by writing Rectangle::print().**
- In general, if the base class is `A` and `B` derives from `A` and `C` derives from `B`, then we select the specific function we want by writing `A::print()` or `B::print()` or `C::print()`.
- Note as always that the above remarks apply only to **public and protected methods.**
- A derived class cannot access the private methods of the parent class.

1.7 Derived class Square: function bodies

```
class Square : public Rectangle {
public:
  Square();
  Square(double e, bool c);
  ~Square();

  double edge() const { return height(); }

  void print() const;

protected:
  bool centered;
};

Square::Square() : Rectangle()
{
  centered = false;
  cout << "Square: default constructor" << endl;
}

Square::Square(double e, bool c) : Rectangle(e, e)
{
  centered = c;
  cout << "Square: non-default constructor" << endl;
}

Square::~Square()
{ cout << "Square: destructor" << endl; }

void Square::print() const {
  if (centered == false) {
    Rectangle::print();                               // call base class method
  }
  else {
    double d = w*0.5;
    cout << "Square::print ";
    cout << "(" << -d << "," << -d << "), ";
    cout << "(" <<  d << "," << -d << "), ";
    cout << "(" <<  d << "," <<  d << "), ";
    cout << "(" << -d << "," <<  d << ") ";
    cout << endl;
  }
}
```

1.8 Derived class Square: main program #1

- Here is a main program to use the class Square. A Square object is instantiated using the default constructor, some methods are executed and the object goes out of scope.

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

class Rectangle;                    // etc  (class declaration)
class Square : public Rectangle;    // etc  (class declaration)

int main()
{
  Square sdef;
  cout << sdef.height() << endl;
  cout << sdef.width() << endl;
  cout << sdef.edge() << endl;
  cout << sdef.area() << endl;
  return 0;
}
```

- Here is the program output:

Rectangle: default constructor
Square: default constructor

```
1                                   // height (default value)
1                                   // width (default value)
1                                   // edge
1                                   // area
```

Square: destructor
Rectangle: destructor

- A line of output is printed whenever a constructor or destructor is called, to keep track of the order of construction and destruction.

1. When the Square object is instantiated, **the Rectangle constructor is called first.**
2. The "implicit Rectangle object" in the Square object is constructed first.
3. **The Square constructor is called next**, to construct the Square part of the object.
4. The order of destructon is the opposite of construction.
5. When the Square object goes out of scope, **the Square destructor is called first.**
6. This destroys the "Square" part of the object.
7. **The Rectangle destructor is called next**, to destroy the "implicit Rectangle object" in the Square object.

1.9 Derived class Square: main program #2

- Here is another main program to use the class Square.
- We employ the non-default constructor, and set the value of centered.
- We instantiate two objects, one with centered==true and one with centered==false.
- We call the method print to demonstrate the difference in the outputs.
- Just for practice, we instantiate the objects dynamically using operator new and delete.

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

class Rectangle;                    // etc  (class declaration)
class Square : public Rectangle;    // etc  (class declaration)

int main()
{
  double len = 3.0;
  Square * ps = new Square(len, true);
  ps->print();
  delete ps;

  ps = new Square(len, false);
  ps->print();
  cout << ps->height() << endl;
  cout << ps->width() << endl;
  cout << ps->edge() << endl;
  cout << ps->area() << endl;
  delete ps;
  return 0;
}
```

- **See next page(s).**

- Here is the program output:

  Rectangle: non-default constructor

  Square: non-default constructor

  Square::print $(-1.5, -1.5)$, $(1.5, -1.5)$, $(1.5, 1.5)$, $(-1.5, 1.5)$ // override

  Square: destructor

  Rectangle: destructor

  Rectangle: non-default constructor

  Square: non-default constructor

  Rectangle::print $(0, 0)$, $(3, 0)$, $(3, 3)$, $(0, 3)$ // base class method

  3 `// height (non-default value)`

  3 `// width (non-default value)`

  3 `// edge`

  9 `// area`

  Square: destructor

  Rectangle: destructor

- The two outputs from the calls to `print()` demonstrate the different behaviors.
- The non-default constructors are invoked, again in the order `Rectangle` first and `Square` next.
- The destructors are called in the opposite order, first `Square` then `Rectangle`.
  1. The call to operator `new` invokes the constructor.
  2. The call to operator `delete` invokes the destructor.
- The height and width are set to the non-default value of 3.0.
- Hence the area is 9.0, and the correct numbers are printed.

2 Big Three: copy, assign, destroy

- The classes `Rectangle` and `Square` do not have any pointer data members and do not allocate memory internally, hence a shallow copy works correctly for them.
- Nevertheless, we can write copy constructors and assignment operators for them.
- The lessons we learn will be sufficient to explain the relevant concepts to write a copy constructor and assignment operator for a derived class.
- Note that we already know how to write the destructor for a derived class. It looks the same as the destructor of a non-derived class.
- Hence this section is really about the "Big Two" not the Big Three.
- First we write the copy constructor and assignment operator for the `Rectangle` base class.
- We print a line of output to screen for educational purposes.

```
Rectangle(const Rectangle &orig)
{
  h = orig.h;
  w = orig.w;
  cout << "Rectangle: copy constructor" << endl;
}

Rectangle& operator= (const Rectangle &rhs)
{
  if (this == &rhs) return *this;
  h = rhs.h;
  w = rhs.w;
  cout << "Rectangle: assignment operator" << endl;
  return *this;
}
```

- **See next page(s).**

2.1 Copy constructor of derived class

- The signature of the copy constructor follows the same rules as every C++ class.

```
Square(const Square& orig);                         // copy constructor
```

- We write the copy constructor of the `Square` class as follows.

```
Square::Square(const Square& orig) : Rectangle(orig)
{
    centered = orig.centered;
    cout << "Square:  copy constructor" << endl;
}
```

- Conceptually, it is a straightforward procedure how to make a copy.
  1. As with the default and non-default constructors, we call the constructor of the base class in the member initialization list.
  2. In this case, we call the copy constructor of the base class.
  3. The `Rectangle` copy constructor **makes a copy of the `Rectangle` part of the object.**
  4. After that, we **copy the data in the `Square` part of the object.**
  5. The above code performs a shallow copy, but the procedure for a deep copy is obvious.

- **However, notice something peculiar:**
  1. The input argument "`const Rectangle &`" of the `Rectangle` copy constructor is (by definition) a reference to a `Rectangle` object.
  2. However, we are passing it an object of type `Square`:

     Rectangle(orig) $\longleftarrow$ **object of type Square**

  3. ***How and why does this work correctly?***

- The answer lies in the "IS-A" concept.
- Every object of type `Square` "IS-A" `Rectangle`.
- Hence the input reference of the `Rectangle` copy constructor **binds to the `Rectangle` part of the input `Square` object.**
  1. In general, a reference to the base class can bind to an object of a derived class.
  2. The reference binds to the "implicit base class object" in the derived class object.
  3. A pointer to the base class can point to the address of an object of a derived class.
  4. The pointer points to the address of the "implicit base class object" in the derived class object.

2.2 Assignment operator of derived class

- The signature of the assignment operator follows the same rules as every C++ class.

```
Square& operator= (const Square& rhs);          // assignment operator
```

- We write the assignment operator of the `Square` class as follows.

```
Square& Square::operator= (const Square& rhs)
{
    if (this == &rhs) return *this;
    Rectangle::operator=(rhs);       // copy Rectangle part of input object
    centered = rhs.centered;
    cout << "Square:  assignment operator" << endl;
    return *this;
}
```

- The assignment procedure is also conceptually straightforward.
  1. First we perform the test for self-assignment.
  2. **Next, we invoke the assignment operator of the base class.**
  3. This **makes an assignment copy of the `Rectangle` part of the input object.**
  4. After that, we **copy the data in the `Square` part of the object.**
  5. The above code performs a shallow copy, but the procedure for a deep copy is obvious.

- **However, notice something peculiar:**
  1. The statement "**Rectangle::operator=(rhs)**" makes a copy of the `Rectangle` part of the input object, but it does not look like the usual syntax for assignment.
  2. In the usual syntax for assignment the "=" sign appears between the operands, e.g. $x = y$.
  3. However, there is nothing on the left hand side.
  4. Also "`rhs`" is enclosed in parentheses "**(rhs)**" like an input to a function call.
  5. **The statement "Rectangle::operator=(rhs)" is a function call.**
  6. The assignment operator is not only an operator, *it is also a class method.*
  7. In the above context, "**Rectangle::operator=**" is employed as a function.
  8. As with the copy constructor, the input to "Rectangle::operator=" **binds to the `Rectangle` part of the `Square` object.**

3 Slicing

- Suppose we have a `Rectangle` object r and a `Square` object s.
- The following assignment statement compiles and runs correctly.

```
Square s(3.0, true);
Rectangle r;
r = s;                      // works
```

- The compiler automatically calls the assignment operator `Rectangle::operator=` and copies (assigns) only the `Rectangle` part of the `Square` object.
- Hence r contains a copy of the `Rectangle` part s.
- The above operation is called **slicing.**
- The compiler "slices off" the non-Rectangle part of s and copies only its `Rectangle` part.
  1. In the above context, only the data members `h,w` are copied:

     ```
     r.h = s.h;
     r.w = s.w;
     ```

  2. The data member `s.centered` is not copied because it is in the `Square` part of s.
- Slicing also occurs if the copy constructor is invoked (e.g. call by value in function calls).

```
Square s(3.0, true);
Rectangle r_copy(s);         // copy constructs only Rectangle part of s
```

- **The converse does not work.**

```
Square s(3.0, true);
Rectangle r;
Square s_copy(r);           // WILL NOT COMPILE
s = r;                      // WILL NOT COMPILE
```

- The compiler does not automatically know how to construct the `Square` part of `s_copy`.
- The compiler does not know what to do to assign the data in the `Square` part of s.

4 References & Pointers: base class and derived class

- **A reference to a base class can bind to an object of a derived class.**
- **A pointer to a base class can point to the address of an object of a derived class.**
- This is a consequence of the "IS-A" concept.
- The reference binds to the base class part of the object of the derived class.
- The pointer points to the address of the base class part of the object of the derived class.
- The following is valid C++ code.

```
Square s(3.0, true);
Rectangle &r_ref = s;           // binds to Rectangle part of s
Rectangle *r_ptr = &s;          // points to address of Rectangle part of s
```

- **As one might guess, the converse does not work.**

```
Rectangle r;
Square &s_ref = r;           // WILL NOT COMPILE
Square *s_ptr = &r;          // WILL NOT COMPILE
```

- The compiler does not know how to bind the `Square` part of `s_ref` using the input r.
- Because r does not have a `Square` part, the compiler cannot set an address for `s_ptr`.

5 Example: main program #3

- The main program and functions below demonstrate copy, assignment and slicing.
- It should be clear at each step what is going on.
- Note that the copy constructor and assignment operator must be added to the `Rectangle` and `Square` classes.

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

class Rectangle;                    // etc  (class declaration)
class Square : public Rectangle;   // etc  (class declaration)

void display_rectangle(Rectangle r)    // call by value
{
  r.print();
}

void display_square(Square s)          // call by value
{
  s.print();
}

int main()
{
  Square s1;
  Square s2(3.0, true);
  s1 = s2;                             // assignment
  s1.print();
  display_square(s1);                  // copy
  display_rectangle(s2);               // slicing (input to function call)

  Rectangle *r_ptr = &s1;
  Rectangle &r_ref = s1;
  Rectangle r_copy(s1);                // copy uses slicing
  Rectangle r;
  r = s1;                              // slicing

  r_ptr->print();                      // print() of Rectangle class
  r_ref.print();                       // print() of Rectangle class
  r_copy.print();                      // print() of Rectangle class
  r.print();                           // print() of Rectangle class
  return 0;
}
```