Classes and Objects in C++: Operator Overloading

CSC 112
Fall 2009
Question:

How can we compare two objects of a given class to see if they are the same?

Is it legal to do:

```cpp
Rectangle r(0,0,3,3);
Rectangle s(0,0,4,4);
cout << (r == s) << endl;
```
OOP: Operator Overloading

Rectangle r(0, 0, 3, 3);
Rectangle s(0, 0, 4, 4);
cout << (r == s) << endl;

Illegal with our current definition of Rectangles (what we had learned up until 11/4).
Essentially, compiler doesn’t know what it means for two Rectangles to be the same.
OOP: Operator Overloading

- Fortunately, C++ allows the programmer to redefine what a built-in C++ operator means within user-defined types.

- Note, one cannot make use of built-in operators in Java!

- As an example, the C++ test for equality is ==
  - int a = 3; int b = 3; if (a == b)
  - Rectangle r(0,0,1,1); Rectangle s(0,0,2,2); if (r == s)
OOP: Operator Overloading

- Must adhere to language defined operator signature when overloading
- Need to implement a definition for the operator given the context of the class the operator is defined in
- Define equality of Rectangles as:
  - Same xPos, yPos start points
  - Same width, same height
// tests whether input rectangle s is equal to
// our rectangle, given definition on previous slide
// signature needs to be in header file as well
bool Rectangle::operator== (const Rectangle& r)
{
    if ((xPosition == r.xPosition) && (yPosition == r.yPosition)
        && (height == r.height) && (width == r.width)) return true;
    else return false;
}

// Checks for equality in x1,y1,h,w
OOP: Operator Overloading

- Shortcut Equality Test
  If actually working with same physical object, then pointers to two objects will be the same.

  Pointer to current object: `this`
  Pointer to other object: `&inputObject`

  Test for same object: `if (this == &inputObject)`

- Potential to save a lot of time if comparing large objects and often compare the same underlying object
OOP: Operator Overloading

// tests whether input rectangle s is equal to our rectangle, given definition on previous slides
bool Rectangle::operator==(const Rectangle& r)
{
    if (this == &s) return true;
    else if ((xPosition == r.xPosition) && (yPosition == r.yPosition) && (height == r.height) && (width == r.width)) return true;
    else return false;
}

// checks if same object first (this pointer)
// then checks for equality in xPosition, yPosition, height, width
A Key Idea:
- Calling `A == B`, where `==` is a function defined for the class, can also be done with this code:
  ```
  A.operator==(B)
  ```
  You are calling the member function `operator==` on object `A` with parameter object `B`.

Fits with our syntax for defining overloaded operators
- Present in header and implementation file
- Scoped as part of object

Holds for most cases! There are some exceptions we’ll see!
OOP: Operator Overloading

- **Assignment:**
  - Object objectOne, objectTwo;
  - objectOne = objectTwo;
  - objectOne’s variables are updated to be equal to objectTwo’s variables

- **Addition:**
  - Object objectOne, objectTwo, objectThree;
  - objectThree = objectOne + objectTwo;
  - objectOne is not changed, even though + is called on it
OOP: Operator Overloading

- Addition Semantics:
  ```
  Object objectOne, objectTwo, objectThree;
  objectThree = objectOne + objectTwo;
  ```

- Addition Implementation:
  - Should generate new object of object type
  - Should update variables of new object to be sum of variables from two input objects (the calling object and the parameter object).
  - Return that new object.
OOP: Operator Overloading

- Not all operators are member functions
- Don’t make sense in the context of working on the current object
OOP: Operator Overloading

- When using non-member functions:
  - Not actually calling member operation on the object, but instead passing object in
  - Can overload other operators this way, such as addition:

```
friend Rectangle operator+ (const Rectangle & r1, const Rectangle & r2);
```
Symmetric Operations

- In some cases, need to use free, non-member functions to allow symmetry of operator.

- Assume scaling operation using * (multiply), where width and height of rectangle are multiplied by a double input value:

  Rectangle r = r * 2;
  Rectangle r = 2 * r;

Would like these to be equivalent.
Symmetric Operations

If * is a member function:

```cpp
Rectangle Rectangle::operator*(double input)
```

Then having both operations work:

```cpp
Rectangle r = r * 2;
Rectangle r = 2 * r;
```

isn’t yet defined in the language! The left-hand-side of the * must be a rectangle object, if only the above function is defined.
Symmetric Operations

- Actually need two functions to support symmetry:
  
  // member or non-member - Rectangle on Left Hand Side
  Rectangle Rectangle::operator*(double input);
  (equivalent to: Rectangle operator*(const Rectangle & &r, double input); )

  // and a non-member function - Rectangle Right Hand Side
  Rectangle operator*(double input, const Rectangle & r);

Friend Functions

- For non-member functions to make use of private member variables or private member functions, they need to be declared as a friend.

- In Rectangle header file, define operator as:

  ```cpp
  friend Rectangle operator +(const Rectangle & r1, const Rectangle & r2);
  (friend is only needed in the .h file, not the .cpp file)
  ```

Meaning that the operator + function can directly read and write all private Rectangle variables and functions.
Friends

- A function is granted friend status from a class, allowing it to read and write private parts of the class.
- Friendship is a one-way function.
  - The friend can read and write variables and functions from the granting class.
  - The granting class does not have any privileged status with the friend just because it made it a friend.