CS170#06.1

Operator Overloading As Methods

Vadim Surov

Outline

- Operator Overloading As Methods
- Overloading For **cout**
- Automatic Conversions
- Side-Effect Operators
- Issues With Operator Overloading

Operator Overloading As Methods

- So far our operator functions are global functions
- They made use of the `GetSeconds()` public method
- If the `GetSeconds()` method did not exist, then none of the functions will work
- One solution is to make these functions member functions of the `StopWatch` class

Operator Overloading As Methods

```
class StopWatch
{
public:
   // Public methods...

   // Operator overloads
   StopWatch operator+(const StopWatch& rhs) const;
   StopWatch operator-(const StopWatch& rhs) const;
   StopWatch operator*(int rhs) const;

private:
   int seconds;
};
```

Operator Overloading As Methods

```
StopWatch StopWatch::operator+(
                  const StopWatch& rhs) const
{
  // Add seconds to this object's seconds
  StopWatch sw(seconds + rhs.seconds);
  return sw;
}
StopWatch StopWatch::operator-(
                  const StopWatch& rhs) const
{
  // Add seconds to this object's seconds
  StopWatch sw(seconds - rhs.seconds);
  return sw;
}
```

Operator Overloading As Methods

- The first argument (that corresponds to the left-hand side operand) is omitted
  - The calling object is assumed to be the left operand
- Most of the previous statements work as before
- But this no longer works:

```
StopWatch sw2 = 2 * sw1;
```

  - Left operand is **int**, not StopWatch
  - So we still need a non-member operator for this case

Overloading For cout

- Instead of Display(), we want to be able to do this:

```
StopWatch sw1;
std::cout << sw1;
```

- We achieve this by overloading the << operator in a non-member function
    - The code is almost identical to Display()

Overloading For cout

```
std::ostream& operator<<(std::ostream& os,
                         const StopWatch& sw){
  ...
  os.fill('0');
  os << std::setw(2) << h << ':';
  os << std::setw(2) << m << ':';
  os << std::setw(2) << s << std::endl;

  // return the reference to ostream
  return os;
}
```

Overloading For cout

- Now doing this:

```
std::cout << sw1;
```

- Is the same as this:

```
operator<<(std::cout, sw1);
```

- Because we return a reference to an ostream object, we can even do this:

```
std::cout << sw1 << sw2 << sw3;
```

Overloading For cout

- What if there is no GetSeconds() public method?
    - Can we declare a non-member function?
    - Can we declare a member function?
- Solution: allow a particular function access to private data by declaring it a **friend**

Overloading For cout

- Use the **friend** keyword in the function prototype in the class declaration:

```
class StopWatch {
public:
   ...
   // friend function
   friend std::ostream& operator<<(
     std::ostream& os, const StopWatch& sw);

private:
   int seconds;
};
```

Automatic Conversions

- Can you do this?

```
StopWatch sw2 = sw1 + 60;
```

- Recall that the C++ compiler performs certain automatic conversions, e.g.,

  3 + 4.1 is 7.1 (double)

  Same as (double) 3 + 4.1

- For our example above, the compiler does this:

```
StopWatch sw2 = sw1 + (StopWatch) 60;
StopWatch sw2 = sw1 + StopWatch(60);
```

Automatic Conversions

- Any constructor that takes one argument is called a *conversion constructor*
  - Implicitly called by the compiler
- Example:

```
StopWatch sw1; // default constructor
sw1 = 60; // sw1 = (StopWatch) 60;
          // sw1 = StopWatch(60);
```

Automatic Conversions

- Another example:

```
void fooSW (const StopWatch & sw)
{
  sw.Display();
}
```

- Calling the function:

```
StopWatch sw(60);
fooSW(sw);
fooSW(60);
```

Automatic conversions

- If you do not want a constructor to perform automatic conversion, use the **explicit** keyword:

```
class StopWatch
{
public:
   // explicit constructor
   explicit StopWatch(int seconds);
private:
   int seconds;
};
```

Automatic conversions

- Now for this example:

```
void fooSW(const StopWatch & sw) {
  sw.Display();
}
```

- Calling the function:

```
StopWatch sw(60);
fooSW(sw); // OK
fooSW(60); // Error
fooSW(StopWatch(60)); // OK
```

Automatic conversions

- However, you cannot do this:

```
StopWatch sw;
int sec = sw; // Error
int sec2 = (int) sw; // Error
```

- The conversion constructor can convert an **int** to a StopWatch, but not a StopWatch to an **int**
- To convert a StopWatch object to an **int**, we can write a function

Automatic conversions

```
class StopWatch {
public:
  // Public methods
  ...
  // conversion to int
  int ToInt(void) const;

private:
  int seconds;
};

int StopWatch::ToInt(void) const {
  return seconds;
}
```

Automatic conversions

- To use this, you must call it explicitly:

```
StopWatch sw(60);
int seconds = sw.ToInt();
std::cout << sw.ToInt();
```

- If you want to give the ability for *implicit* conversion, define a member function using the **operator** keyword

Automatic conversions

```
class StopWatch {
public:
  // Public methods
  ...
  // implicit conversion to int
  operator int(void) const;

private:
  int seconds;
};

StopWatch::operator int(void) const {
  return seconds;
}
```

Automatic conversions

- Now implicit conversion works:

```
StopWatch sw(60);
int seconds = sw;
std::cout << sw;
```

- Notes:
  - General form: **operator** type()
  - It must be a member function
  - No parameter (you may have **void** as parameter)
  - No return type (can't even return **void**)

Automatic conversions

- Be careful when using implicit conversions
  - These conversions are done silently

- Example

```
int array[10];
StopWatch temp1(60);
int temp2 = 0;
array[temp1] = 10; // Uh-oh...
```

Side-effect operators

- Recall that side-effect operators modify the left operand
- Suppose we want to do this:

```
StopWatch sw1(60), sw2(30);
sw1 += sw2; // Now sw1 == 90
```

Side-effect operators

- Comparing with **operator+**:

```
class StopWatch {
public:
   // Public methods...
   // overload for sw1 + sw2
   StopWatch operator+(const StopWatch& rhs) const;

   // overload for sw1 += sw2
   StopWatch& operator+=(const StopWatch& rhs);

private:
   int seconds;
};
```

Side-effect operators

- Comparing with **operator+**:

```
StopWatch StopWatch::operator+(
               const StopWatch& rhs) const {
  // create a new object from both operands
  StopWatch sw(seconds + rhs.seconds);
  return sw;
}

StopWatch& StopWatch::operator+=(
                    const StopWatch& rhs) {
  // modify this object directly
  seconds += rhs.seconds;
  return *this;
}
```

Side-effect operators

- Notes:
  - The method is not marked **const** because it changes the calling object
  - It returns a reference because we are not creating a new object
  - Since **this** is a pointer to the calling object, ***this** is the object itself
  - Returning the object allows the following:

```
sw1 += sw2 += sw3;
```

Side-effect operators

- How do you overload the ++ operator?
- Prefix:

```
class StopWatch {
public:
   // Public methods...
   // overload for prefix ++
   StopWatch& operator++(void);
private:
   int seconds;
};
StopWatch& StopWatch::operator++(void) {
   seconds++;
   return *this;
}
```

Side-effect operators

- The postfix version of ++ has an **int** parameter:

```
class StopWatch {
public:
  // Public methods...
  // overload for postfix ++
  StopWatch operator++(int);
private:
  int seconds;
};
StopWatch StopWatch::operator++(int) {
  StopWatch sw(seconds);
  seconds++;
  return sw;
}
```

Side-effect operators

- The int parameter for postfix++ (and postfix--) is not used; it is just to indicate postfix rather than prefix
- Because the postfix version requires the creation of a temporary object, in general prefix increment is more efficient than postfix
- Bonus question:
    - With these examples, sw1 = (++sw2)++; is legal, and so is (sw1 + sw2) = 30; How do you prevent this?

Issues with operator overloading

- With operator overloading, in general there are 3 options. E.g., when overloading a binary operator:
  - Member function (one operand, implicit this)
  - Non-member friend function (two operands)
    - friend has access to private data
  - Non-member, non-friend function
    - data access via public methods

Issues with operator overloading

- At least one operand must be a user-defined type (you can't overload built-in types)

```
// Illegal - overloading int addition
int operator+(int lhs, int rhs);
```

- You cannot violate C++ rules for the operator you overload
    - Number of operands
    - Precedence
    - Associativity
    - E.g., you cannot overload modulo (%) to take one operand

Issues with operator overloading

- You cannot create new operator symbols. E.g., **operator@** is not allowed because @ is not a C++ operator
- The following operators can *only* be overloaded as member functions (others can be overloaded as non-member functions):

=	Assignment operator
()	Function call operator
[]	Subscripting operator
->	Class member access by pointer operator

Issues with operator overloading

- You cannot overload the following operators:

::	Scope resolution operator
.*	Pointer-to-member operator
.	Membership operator
?:	Conditional operator
sizeof	sizeof operator
typeid	RTTI operator
const_cast	A typecast operator
dynamic_cast	A typecast operator
reinterpret_cast	A typecast operator
static_cast	A typecast operator

Summary

- Operator overloading allows user-defined types to be used as operands for operators like in-built types
- This is done by defining an **operator** function using the operator keyword
- In a non-member, non-friend function, the first argument corresponds to the left operand while the second argument corresponds to the right operand

Summary

- For operator commutativity such that a built-in type is the left operand, write an additional function
- In a member function, the left operand is the calling object while right operand corresponds to the argument
- A friend function is marked with the **friend** keyword in the prototype; it has access to the private data of the class

Summary

- The function to overload << for std::cout should return std::ostream& and have std::ostream& as the first parameter
- Constructors with only one argument are used with automatic conversions
  - To override this behaviour, mark the constructor as **explicit**
- To allow automatic conversion of a class to an in-built type, declare an **operator** *type* function

Summary

- Side-effect operator overloads usually return a (constant?) reference to the class
- The postfix ++ overloading function is denoted by an unused int parameter
- The =, (), [] and -> operators can only be overloaded as member functions