Operator Overloading, Friends, and Conversion

The Plan For Tonight
- Reading: 445-501 (Chapter 11)
- Operator overloading [445-458]
- Friend functions [458-460]
- Overloading the insertion operator [461-467]
- Overloading the inc/decrement operators [NIB]
- Classes and type-conversions [483-499]
- Homework 6, 7

Operator Overloading I
- Operator overloading is like function overloading, but with some restrictions
- Example: Overload1.cpp
  - The EngDist class: feet and inches
  - The working/default constructor
    - Use of default arguments
  - A conversion constructor
  - The print() method
Operator Overloading II

<table>
<thead>
<tr>
<th>Writing a method to add two EngDist objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>- What should it take as an argument?</td>
</tr>
<tr>
<td>- What should it return?</td>
</tr>
<tr>
<td>Here's a solution</td>
</tr>
</tbody>
</table>

EngDist EngDist::add(const EngDist & rhs) {
    EngDist temp(feet + rhs.feet, inches + rhs.inches);
    return temp;
}

Operator Overloading III

<table>
<thead>
<tr>
<th>Using the add() method [Overload01.cpp]</th>
</tr>
</thead>
<tbody>
<tr>
<td>EngDist a(3, 5.25), b(7.25);</td>
</tr>
<tr>
<td>a.add(b); // What happens here?</td>
</tr>
<tr>
<td>EngDist c = a.add(b); // How about here?</td>
</tr>
<tr>
<td>EngDist d = a.add(5.75); // OK?</td>
</tr>
<tr>
<td>EngDist e = a.add(3); // OK?</td>
</tr>
<tr>
<td>EngDist f = a.add(3L); // OK?</td>
</tr>
</tbody>
</table>

In C++ we can change method name like this:

return-type operator#(arg-list);

Operator Overloading IV

<table>
<thead>
<tr>
<th>The new &quot;add&quot; function [Overload02.cpp]</th>
</tr>
</thead>
<tbody>
<tr>
<td>EngDist operator+(const EngDist &amp; rhs) const</td>
</tr>
</tbody>
</table>

Calling the new function

EngDist a(3, 5.25), b(7.25);
EngDist c = a.operator+(b);
EngDist d = a.operator+(5.75);

Not much of an improvement, is it?
Operator Overloading V

Fortunately, C++ provides some syntax sugar
- Instead of writing this [Overload03.cpp]
  - EngDist c = a.operator+(b);
- C++ lets us eliminate the .operator and parens
  - EngDist c = a.operator+(b);
- Leaving us with this, which is much more readable
  - EngDist c = a + b;

Friend Functions I

- What are friend functions?
  - A "free" function with access to private class data
- Why would you want them?
  - Often used for operator overloading
  - Used to allow use of cout with class objects
  - Allow interaction of "related" classes
- Somewhat controversial in some circumstances

Friend Functions II

- Why use friend functions for overloaded operators?
  - Using member functions, this is OK
    - EngDist c = a + 5.25;
  - Remember, this "really" translates to:
    - EngDist c = a.operator+( 5.25 );
  - What happens when you do this?
    - EngDist c = 5.25 + a;
Friend Functions III

- It doesn't work because there is no function with this signature (which is what is requested)
  - `double.operator+(EngDist&);`

- Member function requires object on left side
  - Since friend function is a "free" function, it doesn't
    - Friend functions can access private class data
    - But they don't automatically get a this pointer
    - Instead, you have to pass an object as an argument

Friend Functions IV

- Creating a friend function is a two-step process
  - First, you must declare the function in "friend" class
    - Class must grant friendship; cannot be "taken"
    - You must pass all objects as arguments
    - There is no implicit this pointer passed
    - For a binary operator, pass both left and right args
  - Addition operator in the EngDist class [Overload04.cpp]
    - `class EngDist {
        friend EngDist add(const EngDist& lhs, const EngDist& rhs);
    };

Friend Functions V

- Once declared, define function outside class
  - Write function outside class [not a member]:
    - `EngDist operator+( const EngDist& lhs, const EngDist& rhs) {
        return EngDist(lhs.feet + rhs.feet,
                       lhs.inches + rhs.inches);
    }

- Single-arg constructor makes this work
  - Auto-converts 5.25 to EngDist object lhs
An "inserter" lets you use cout with your objects

Wouldn't it be nicer to do this:

```cpp
EngDist a(2, 7.5);
cout << "a = " << a << endl;
```

Instead of this:

```cpp
EngDist a(2, 7.5);
cout << "a = "; a.print(); cout << "n";
```

Can't be EngDist member; object on left is cout

To write an inserter that works with cout you:

1. Write a "free" function named `operator<<(ostream& out, const object&)`
2. The function returns an `ostream&`
3. The first argument is an `ostream&`
4. The second argument is a `const object&`
   - Assumes that object is your type [EngDist]
5. Use the first argument like you would cout
6. Return the first argument when done

General form of an inserter function

```cpp
ostream & operator<<(ostream& out, const EngDist & a)
{
    // Stuff
    return out;
}
```

Not a member function

- Usually a friend so that it has access to data
Writing an Inserter IV

- Let's change EngDist::print() to an inserter
- Change the class declaration like this:

```cpp
class EngDist
{
void print() const;
friend ostream&
operator<<(ostream&, const EngDist&);
};
```

Writing an Inserter V

- Make these changes to the print() definition

```cpp
void EngDist::print() const
ostream&
operator<<(ostream& out, const EngDist& d)
{
    cout << d.feet << "' " << d.inches << ""
    return out;
}
```

Writing an Extractor I

- Can use cin [or any istream] to read your objects
- General form is very similar to inserter
- Use the istream class, not ostream
- Make sure your object & is not constant

```cpp
istream&
operator>>(istream& in, obj& o)
{
    // Use in to populate your fields
    return in;
}
```
Writing an Extractor II

- Here's a simple extractor for the EngDist class
  - Assumes input is in the form 9' 6.25"

```cpp
istream& operator>>(istream& in, EngDist& d)
{
    char x;
    in >> d.feet >> x >> d.inches >> x;
    return in;
}
```

- A better solution would validate data

Operator Overloading Rules

- Operator overloading rules [page 453-454]
  - Can only overload user-defined classes
  - Can't overload for built-in types like int, double
  - Cannot change the precedence or arity
    - Cannot use ^ as a unary operator, etc.
  - Cannot make up new operators like **
  - All operators are inherited except for =
  - Default arguments are not allowed
  - Some operators can't be overloaded [page 453]

Designing Overloaded Operators

- How should your operators act?
  - Think about how operator is used
  - Use the principle of "least astonishment"
  - Example: What does a * b mean?
    - It really depends on what a and b represent
    - For EngDist, use to get "square footage" of a rectangle
  - Look at these expressions. Do they make sense?
    - 3' 5" * 2 == 6' 10"
    - 10' 6" / 2 == 5' 3"
    - 10' 6" / 5' 3" == 2.0
Overloaded Overloaded Ops I

- Notice that we have two meanings for divide
  - When dividing an EngDist by a double we mean:
    » "What is 50% of distance a?"
    » The result is an EngDist object
  - When dividing one EngDist by another, we mean:
    » "What is the relationship [as a percent] between distance a and distance b?"
    » The result is a floating-point number
  - We need different versions of the division operator

Overloaded Overloaded Ops II

- Overload operator/(), by using different args
  - Both should be member functions, not friends
  - Use const so your objects act like built-in versions
    » Don't want users doing this kind of bizarre thing

Overloading a unary operator

- Unary ops [as members] require no arguments
  - Will normally write as a member function
  - Example: Unary minus [Overload06.cpp]
    » What should it return? A negative EngDist object
      - const EngDist EngDist::operator - ()
        {
          return EngDist(-(feet * 12 + inches));
        }
  - Another (better?) possibility is to have a "sign" member
Overloading ++ I

What should the increment operator do?
- Add a foot or add an inch?
  » You have to decide -- no fixed answers
- What does the increment operator return?
  » A const object of the same type
  » For prefix, we can return a const EngDist&
- What happens to the object itself?
  » It changes the object
- Example: Overload07.cpp

Overloading ++ II

Looks like we have a problem with postfix ++
- How does the compiler tell prefix/postfix?
- You use a dummy int argument to define postfix:
  - const EngDist& operator++();    // prefix
  - const EngDist operator++(int); // postfix
- When you write a prefix EngDist expression, the compiler calls the first function. When you write a postfix expression, it calls the second. The dummy argument is never actually used [by you].

Overloading ++ III

Note that the semantics of postfix & prefix differ
- Prefix changes the object then returns it
- Postfix returns the object as it was before modification

Here’s how to implement postfix increment
- Write the function in terms of prefix (Overload08.cpp)
  » 1) Construct a temporary object using *this
  » 2) Increment *this using prefix
  » 3) Return the temporary object
Homework

- Homework 5: Overloaded Operators
  - Due Sunday February 29, [Grade early Mon AM]
  - Add to the Fraction class
    » Overloaded operators for add, subtract, multiply, divide
    » Overloaded insertion operator [replacing print()]
    » Overloaded extraction operator [replacing read()]
  - Use Fraction03.h, hmwk05.cpp without changes
  - Define methods in Fraction03.cpp
  - Submit only Fraction03.cpp [Double-check]

Conversions I

- Conversion: transferring value while changing type
  - Conversion between primitives is automatic
    - double d; int i;
      i = 3.14; d = 7;
  - User-defined conversions are not automatic (implicit)
    - Dist a(7,3.2); double b = 7.25;
      a = 3.14; // Not done automatically
      b = a;    // Not done automatically

Conversions II

- For objects, all conversions must be programmed
  - Three different kinds of programmable conversions
  - 1) Converting from a primitive type to an object type
    - EngDist d; d = 25;
  - 2) Converting from an object type to a primitive type
    - EngDist x(5, 2.5); int i; i = x;
  - 3) Converting between two object types
    - Fraction f(1, 5); EngDist d; d = f;
Conversions III

- How do you convert from a primitive to an object?
  - Use a single-argument constructor [Conversion01.cpp]
    ```cpp
    EngDist(double in);
    ```
  - Allows you to use a double when an EnDist is expected
    ```cpp
    void print(EngDist & d)
    {
      cout << d << endl;
    }
    print(325.25);  // Calls conversion constructor
    ```

Conversions IV

- How do you convert an object to a primitive?
  - Use a special conversion member function
  - General syntax is:
    ```cpp
    operator type() { return value; }
    ```
  - Note there is no return type like other operators
- Converts from an EngDist to an int [Conversions02.cpp]
    ```cpp
    operator int() { return inches + feet * 12; }
    ```

Conversions V

- Converting between different object types
  - Assume we have the following MetDist class:
    ```cpp
    class MetDist { long meters, double cm; ... }
    ```
  - How can I do this? [Conversions03.cpp]
    ```cpp
    MetDist met(30, 99.3), met2;
    EngDist eng(5, 7.5), eng2;
    eng2 = met;
    eng2 = eng;
    ```
Method 1: Conversion constructor

- Pass a `MetDist` to a single-arg constructor
  ```cpp
  EngDist(const MetDist & m)
  {
    inches = m.meters * 254 + m.cm * 2.54;
    feet = inches / 12;
    inches = inches - ft * 12;
  }
  ``

- `MetDist` class must grant friendship to `EngDist`:
  ```cpp
  friend class EngDist;
  ```

Method 2: Write a conversion member function

- Does not require friend access or accessors
- Only uses public interface
  ```cpp
  EngDist::operator MetDist()
  {
    return MetDist(feet *.3048, inches * 2.54);
  }
  ``

- Will need a forward declaration before `MetDist` class

Conversions in C++ are "silent" (implicit)

- Often (usually) you don't want silent conversion
- Can use named function instead of operator `type()`
  ```cpp
  const char * c_str() { }
  ```

- Can use keyword `explicit` for 1-arg constructors
  ```cpp
  explicit Metric(int n) { … }
  Metric m(10); // OK
  cout << (3 * m) << endl; // Causes error
  ```
Homework

- Homework
  - Due Sunday February 29, [Grade early Mon AM]
  - Homework 6: RPN Expressions
    - Use istringstream class to tokenize RPN expression
    - Use Stack class to evaluate expression
    - Submit hmwk06.cpp

- Exam 1 - Next Monday
  - Chapters 10 & 11 + Lecture