CSCI 104
Operator Overloading

Mark Redekopp
David Kempe

Function Overloading

- What makes up a signature (uniqueness) of a function
  - name
  - number and type of arguments
- No two functions are allowed to have the same signature; the following 5 functions are unique and allowable…
  - void f1(int); void f1(double); void f1(List<int>&);
  - void f1(int, int); void f1(double, int);

- We say that "f1" is overloaded 5 times

Operator Overloading

- C/C++ defines operators (+,*,-,==,etc.) that work with basic data types like int, char, double, etc.
- C/C++ has no clue what classes we'll define and what those operators would mean for these yet-to-be-defined classes
  - ```
    class complex {
      public:
       double real, imaginary;
    };
    ```
  - ```
    Complex c1,c2,c3;
    // should add component-wise
    c3 = c1 + c2;
    ```
  - ```
    class List {
     ...
    };
    List l1,l2;
    l1 = l1 + l2;  // should concatenate
                   // l2 items to l1
    ```
- **We can write custom functions to tell the compiler what to do when we use these operators! Let us learn how…**

user.h

```
class User{
 public:
   User(string n);  // Constructor
   string get_name();
 private:
   int id_;
   string name_;
};
```

user.cpp

```
#include "user.h"
User::User(string n) {
  name_ = n;
}
string User::get_name(){
  return name_;
}
```

user_test.cpp

```
#include<iostream>
#include "user.h"

int main(int argc, char *argv[]) {
  User u1("Bill"), u2("Jane");
  // see if same username
  // Option 1:
  if(u1 == u2) cout << "Same";

  // Option 2:
  if(u1.get_name() == u2.get_name())
    {   cout << "Same" << endl; }
  return 0:
  }
```
```

Two Approaches

- There are two ways to specify an operator overload function
  - Global level function (not a member of any class)
  - As a member function of the class on which it will operate
- Which should we choose?
  - It depends on the left-hand side operand (e.g. `string + int` or `iostream + Complex`)

Method 1: Global Functions

- Can define global functions with name "operator{+-…}" taking two arguments
  - LHS = Left Hand side is 1st arg
  - RTH = Right Hand side is 2nd arg
- When compiler encounters an operator with objects of specific types it will look for an "operator" function to match and call it

```
int main()
{
  int hour = 9;
  string suffix = "p.m.";

  string time = hour + suffix;
  // WON'T COMPILE…doesn't know how to
  //  add an int and a string
  return 0;
}
```

```
string operator+(int time, string suf)
{
  stringstream ss;
  ss << time << suf;
  return ss.str();
}
int main()
{
  int hour = 9;
  string suffix = "p.m.";


  string time = hour + suffix;
  // WILL COMPILE TO:
  // string time = operator+(hour, suffix);

  return 0;
}
```

Method 2: Class Members

- C++ allows users to write functions that define what an operator should do for a class
  - Binary operators: +, -, *, /, ++, --
  - Comparison operators: ==, !=, <, >, <=, >=
  - Assignment: =, +=, -=, *=, /=, etc.
  - I/O stream operators: <<, >>
- Function name starts with 'operator' and then the actual operator
- Left hand side is the implied object for which the member function is called
- Right hand side is the argument

```
class Complex
{
 public:
  Complex(int r, int i);
  ~Complex();
  Complex operator+(const Complex &rhs);

 private;
  int real, imag;
};

Complex Complex::operator+(const Complex &rhs)
{
   Complex temp;
   temp.real = real + rhs.real;
   temp.imag = imag + rhs.imag;
   return temp;
}

int main()
{
  Complex c1(2,3);
  Complex c2(4,5);
  Complex c3 = c1 + c2;
  // Same as c3 = c1.operator+(c2);
  cout << c3.real << "," << c3.imag << endl;
  // can overload '<<' so we can write:
  // cout << c3 << endl;
  return 0;
}
```

Binary Operator Overloading

- For binary operators, do the operation on a new object's data members and return that object
  - Don’t want to affect the input operands data members
    - Difference between: x = y + z; vs. x = x + z;
- Normal order of operations and associativity apply (can’t be changed)
- Can overload each operator with various RHS types…
  - See next slide

Binary Operator Overloading

```
class Complex
{
 public:
  Complex(int r, int i);
  ~Complex()
  Complex operator+(const Complex &rhs);
  Complex operator+(int real);
 private:
  int real, imag;
};
Complex Complex::operator+(const Complex &rhs)
{
   Complex temp;
   temp.real = real + rhs.real;
   temp.imag = imag + rhs.imag;
   return temp;
}

Complex Complex::operator+( int real )
{
   Complex temp = *this;
   temp.real += real;
   return temp;
}
```

No special code is needed to add 3 or more operands. The compiler chains multiple calls to the binary operator in sequence.

```
int main()
{
 Complex c1(2,3), c2(4,5), c3(6,7);

  Complex c4 = c1 + c2 + c3;
  // (c1 + c2) + c3
  // c4 = c1.operator+(c2).operator+(c3)
  //    = anonymous-ret-val.operator+(c3)


  c3 = c1 + c2;
  c3 = c3 + 5;

}
```

Adding different types (`Complex + Complex` vs. `Complex + int`) requires different overloads

Relational Operator Overloading

- Can overload
  ==, !=, <, <=, >, >=
- Should return bool

```
class Complex
{
 public:
  Complex(int r, int i);
  ~Complex();
  Complex operator+(const Complex &rhs);
  bool operator==(const Complex &rhs);
  int real, imag;
};

bool Complex::operator==(const Complex &rhs)
{
  return (real == rhs.real && imag == rhs.imag);
}

int main()
{
  Complex c1(2,3);
  Complex c2(4,5);
  // equiv. to c1.operator==(c2);
  if(c1 == c2)
    cout << "C1 & C2 are equal!" << endl;

  return 0;
}
```

Nothing will be displayed

Practice On Own

- In the online exercises, add the following operators to your Str class
  - operator[]
  - operator==(const Str& rhs);
  - If time do these as well but if you test them they may not work…more on this later!
  - operator+(const Str& rhs);
  - operator+(const char* rhs);

Non-Member Functions

- What if the user changes the order?
  - int on LHS & Complex on RHS
  - No match to a member function b/c to call a member function the LHS has to be an instance of that class
- We can define a non-member function (good old regular function) that takes in two parameters (both the LHS & RHS)
  - May need to declare it as a friend

```
int main()
{
  Complex c1(2,3);
  Complex c2(4,5);
  Complex c3 = 5 + c1;
              // ?? 5.operator+(c1) ??
              // ?? int.operator+(c1) ??
              // there is no int class we can
              //  change or write

  return 0;
}
```

Doesn't work without a new operator+ overload

```
Complex operator+(const int& lhs, const Complex &rhs)
{
  Complex temp;
  temp.real = lhs + rhs.real;   temp.imag = rhs.imag;
  return temp;
}
int main()
{
  Complex c1(2,3);
  Complex c2(4,5);
  Complex c3 = 5 + c1;   // Calls operator+(5,c1)
  return 0;
}
```

Still a problem with this code
Can operator+(…) access Complex's private data?

Friend Functions

- A friend function is a function that is not a member of the class but has access to the private data members of instances of that class
- Put keyword '**friend**' in function prototype in class definition
- Don't add scope to function definition

```
class Silly
{
  public:
    Silly(int d) { dat = d };
    friend int inc_my_data(Silly &s);
  private:
    int dat;
};

// don't put Silly:: in front of inc_my_data(...)
//  since it isn't a member of Silly
int inc_my_data(Silly &a)
{
   s.dat++;
   return s.dat;
}

int main()
{
   Silly cat(5);
   //cat.dat = 8
   // WON'T COMPILE since dat is private

   int x = inc_my_data(cat);
   cout << x << endl;
}
```

Notice inc_my_data is NOT a member function of Silly. It's a global scope function but it now can access the private class members.

Non-Member Functions

- Revisiting the previous problem

```
class Complex
{
 public:
  Complex(int r, int i);
  ~Complex();
  // this is not a member function
  friend Complex operator+(const int&, const Complex& );
 private:
  int real, imag;
};

Complex operator+(const int& lhs, const Complex &rhs)
{
  Complex temp;
  temp.real = lhs + rhs.real;   temp.imag = rhs.imag;
  return temp;
}
int main()
{
  Complex c1(2,3);
  Complex c2(4,5);
  Complex c3 = 5 + c1;   // Calls operator+(5,c1)
  return 0;
}
```

Now things work!

Why Friend Functions?

- Can I do the following?
- error: no match for 'operator<<' in 'std::cout << c1'
- /usr/include/c++/4.4/ostream:108: note: candidates are: /usr/include/c++/4.4/ostream:165: note: std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT, _Traits>::operator<<(**long int**) [with _CharT = char, _Traits = std::char_traits<char>]
- /usr/include/c++/4.4/ostream:169: note: std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT, _Traits>::operator<<(**long unsigned int**) [with _CharT = char, _Traits = std::char_traits<char>]
- /usr/include/c++/4.4/ostream:173: note: std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT, _Traits>::operator<<(**bool**) [with _CharT = char, _Traits = std::char_traits<char>]
- /usr/include/c++/4.4/bits/ostream.tcc:91: note: std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT, _Traits>::operator<<(**short int**) [with _CharT = char, _Traits = std::char_traits<char>]

```
class Complex
{
 public:
   Complex(int r, int i);
   ~Complex();
   Complex operator+(const Complex &rhs);
 private:
   int real, imag;
};

int main()
{
   Complex c1(2,3);
   cout << c1;  // equiv. to cout.operator<<(c1);
   cout << endl;
   return 0;
}
```

Why Friend Functions?

- cout is an object of type 'ostream'
- << is just an operator
- But we call it with 'cout' on the LHS which would make "operator<<" a member function of class ostream
- Ostream class can't define these member functions to print out user defined classes because they haven't been created
- Similarly, ostream class doesn't have access to private members of Complex

```
class Complex
{
 public:
  Complex(int r, int i);
  ~Complex();
  Complex operator+(const Complex &rhs);
 private:
  int real, imag;
};

int main()
{
  Complex c1(2,3);
  cout << "c1 = " << c1;
  // cout.operator<<("c1 = ").operator<<(c1);

  // ostream::operator<<(char *str);
  // ostream::operator<<(Complex &src);

  cout << endl;
  return 0;
}
```

Ostream Overloading

- Can define operator functions as friend functions
- LHS is 1st arg.
- RHS is 2nd arg.
- Use friend function so LHS can be different type but still access private data
- Return the ostream& (i.e. os which is really cout) so you can chain calls to '<<' and because cout/os object has changed

```
class Complex
{
 public:
  Complex(int r, int i);
  ~Complex();
  Complex operator+(const Complex &rhs);
  friend ostream& operator<<(ostream&, const Complex &c);
 private:
  int real, imag;
};

ostream& operator<<(ostream &os, const Complex &c)
{
  os << c.real << “,“ << c.imag << “j”;
  //cout.operater<<(c.real).operator<<(“,”).operator<<...
  return os;
}

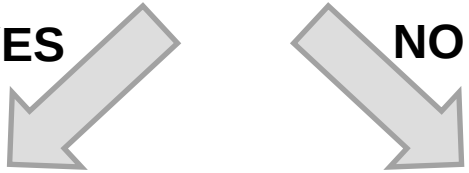
int main()
{
  Complex c1(2,3), c2(4,5);
  cout << c1 << c2;
  // operator<<(cout, c1);
  cout << endl;
  return 0;
}
```

Template for adding ostream capabilities:
friend ostream& operator<<(ostream &os, const T &rhs);
(where T is your user defined type)

Member or Friend?

Should I make my operator overload be a member of a class, C1?

Ask yourself: *Is the LHS an instance of C1?*




```
C1 objA;
objA << objB  // or
objA + int
```

YES the operator overload function can be a **member function** of the C1 class since it will be translate to objA.operator<<(…)

```
C1 objA;
objB << objA // or
int + objA
```

NO the operator overload function should be a **global level (maybe friend) function** such as operator<<(cout, objA). It cannot be a member function since it will be translate to objB.operator<<(…).

Summary

- If the left hand side of the operator is an instance of that class
  - Make the operator a member function of a class…
  - The member function should only take in one argument which is the RHS object
- If the left hand side of the operator is an instance of a different class
  - Make the operator a friend function of a class…
  - This function requires two arguments, first is the LHS object and second is the RHS object