CSCI 104
Operator Overloading

Mark Redekopp
David Kempe
Get the Example Code

- Download the code
  - $ wget http://ee.usc.edu/~redekopp/cs104/str_ops.tar
  - $ tar xvf str_ops.tar
  - $ wget http://ee.usc.edu/~redekopp/cs104/complex.tar
  - $ tar xvf complex.tar

- Str should mimic the C++ string class
  - Properly handle memory allocation
  - Let you treat it like an array where you can do '[i]' indexing
  - Let you do comparison on string objects with '==' and other operators, etc.

- Complex should mimic a complex number
List/Array Indexing

- Arrays and vectors allow indexing using square brackets: [ ]
  - E.g. my_list[i] equivalent to my_list.get(i)
- It would be nice to allow that indexing notation for our List class
- But if we just try it won't compile...How does the compiler know what to do when it sees a List object followed by square brackets
- Enter C++ operator overloading
  - Allows us to write our own functions that will be "tied" to and called when a symbolic operator (+, -, *, [ ]) is used

```cpp
#ifndef LLISTINT_H
#define LLISTINT_H
class LListInt{
    public:
        LList();  // Constructor
        ~LList();  // Destructor
        int& get(int loc);
    ...
    private:
        Item* head_;
};
#endif

int main()
{
    LListInt my_list();
    my_list.push_back(5);
    my_list.push_back(7);
    cout << my_list.get(0) << endl;
    cout << my_list[0] << endl;
    return 0;
}
```
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 3 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;
    c3 = c1 + c2; // should add component-wise
  – Class List {
    ...
  }
  – List l1,l2;
    l1 = l1 + l2; // should concatenate l2 items to l1

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

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

#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;
}
```
Operator Overloading w/ Global Functions

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

```cpp
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;
}
```
Operator Overloading for Classes

- 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

```cpp
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
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;
}

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;
}
Relational Operator Overloading

- Can overload `==, !=, <, <=, >, >=`
- Return bool

```cpp
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

• 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

```cpp
def 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

```cpp
Complex operator+(const int& lhs, const Complex &rhs)
{
    Complex temp;
    temp.real = lhs + rhs.real;    temp.imag = rhs.imag;
    return temp;
}
def 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

```cpp
class Dummy
{
  public:
    Dummy(int d) { dat = d ;
      friend int inc_my_data(Dummy &dum);
  private:
    int dat;
};

// don't put Dummy:: in front of inc_my_data(...) 
int inc_my_data(Dummy &dum)
{
    dum.dat++; 
    return dum.dat;
}

int main()
{
    Dummy dumb(5);
    dumb.dat = 8;  // WON'T COMPIL
    int x = inc_my_data(dumb);
    cout<< x << endl;
}
```
Non-Member Functions

• Revisiting the previous problem

```cpp
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>]

```cpp
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
- Ostrem 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

```cpp
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 1\textsuperscript{st} arg.
  - RHS is 2\textsuperscript{nd} 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

```cpp
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.operator<<(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 ostringstream capabilities:
```
friend ostream& operator<<(ostream &os, const T &rhs);
(where T is your user defined type)
```
Summary

• Make the operator a member function of a class...
  – IF the left hand side of the operator is an instance of that class
  – The member function should only take in one argument which is the RHS object

• Make the operator a friend function of a class if...
  – IF the left hand side of the operator is an instance of another class and right hand side is an instance of the class
  – This function requires two arguments, first is the LHS object and second is the RHS object
Practice

• Add an ostream operator ('<<') to your Str class
Exercises For Home

• Write a '[]' operator member function for your List class
  – Have it throw an exception if the index is out of bounds

• Write an '==' operator to check if two lists have exactly the same contents in the exactly the same order

• Write a '+' operator to append one list to the end of another

```cpp
#include <iostream>
#include "listint.h"

using namespace std;

int main()
{
    List<int> m1, m2;
    m1.push_back(5);
    m2.push_back(5);
    if(m1 == m2){
        cout << "Should print!";
    }
    cout << "0-th item is " << m1[0];
    cout << endl;
    m1[0] = 7;
    if(m1 == m2){
        cout << "Should not print!"; << endl;
    }
    return 0;
}