CSCI 104
Operator Overloading & Copy Semantics

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
#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;
}
```

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

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

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

User::User(string n) {
    name_ = n;
}
string User::get_name() {
    return name_
}
```
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

```c++
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;
}
```

```c++
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
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;
}

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
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

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

```cpp
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.

```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 COMPILE
  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 ostream capabilities:

```cpp
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 you 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;
}
```
Copy constructors and assignment operators

COPY SEMANTICS
Get the Code

• On your VM run the command:
  – `wget http://ee.usc.edu/~redekopp/cs104/copycon.cpp`
this Pointer

- How do member functions know which object’s data to be operating on?
- d1 is implicitly passed via a special pointer call the ‘this’ pointer

#include<iostream>
#include “deck.h”

int main(int argc, char *argv[]) { 
    Deck d1, d2;
    d1.shuffle();
}

#include<iostream>
#include “deck.h”

void Deck::shuffle() {
    cut(); // calls cut()
    // for this object
    for(i=0; i < 52; i++){
        int r = rand() % (52-i);
        int temp = cards[r];
        cards[r] = cards[i];
        cards[i] = temp;
    }
}

#include<iostream>
#include “deck.h”

int main() { Deck d1;
    d1.shuffle();
}

void Deck::shuffle(Deck *this) {
    this->cut(); // calls cut()
    // for this object
    for(i=0; i < 52; i++){
        int r = rand() % (52-i);
        int temp = this->cards[r];
        this->cards[r] = this->cards[i];
        this->cards[i] = temp;
    }
}
Another Use of 'this'

- This can be used to resolve scoping issues with similar named variables.

```cpp
class Student {
public:
    Student(string name, int id, double gpa);

    ~Student();  // Destructor
private:
    string name;  // Destructor
    int id;
    double gpa;
};

Student::Student(string name, int id, double gpa)
{ // which is the member and which is the arg?
    name = name; id = id; gpa = gpa;
}

Student::Student(string name, int id, double gpa)
{ // Now it's clear
    this->name = name;
    this->id = id;
    this->gpa = gpa;
}
```
Struct/Class Assignment

• Assigning one struct or class object to another will perform an element by element copy of the source struct/class to the destination struct/class

```cpp
#include<iostream>
using namespace std;

enum {CS, CECS };

struct student {
    char name[80];
    int id;
    int major;
};

int main(int argc, char *argv[])
{
    student s1,s2;
    strncpy(s1.name,"Bill",80);
    s1.id = 5; s1.major = CS;
    s2 = s1;
    return 0;
}
```
Multiple Constructors

• Can have multiple constructors with different argument lists

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

int main()
{
    Student s1; // calls Constructor 1
    string myname;
    cin >> myname;
    s1.set_name(myname);
    s1.set_id(214952);
    s1.set_gpa(3.67);

    Student s2(myname, 32421, 4.0); // calls Constructor 2
}
```

```cpp
class Student {
    public:
        Student();   // Constructor 1
        Student(string name, int id, double gpa); // Constructor 2
        ~Student();  // Destructor
    string get_name();
    int get_id();
    double get_gpa();
    void set_name(string name);
    void set_id(int id);
    void set_gpa(double gpa);

    private:
        string _name;
        int _id;
        double _gpa;
}
```

```cpp
Student::Student() {
    _name = "", _id = 0; _gpa = 2.0;
}

Student::Student(string name, int id, double gpa) {
    _name = name; _id = id; _gpa = gpa;
}
```
Copy Constructors

• Write a prototype for the constructor that would want to be called by the red line of code

• Realm of Reasonable Answers:
  – Complex(Complex)
    • We will see that this can't be right...
  – Complex(Complex &)
  – Complex(const Complex &)

• We want a constructor that will build a new Complex object (c3) by making a copy of another (c1)
Assignment & Copy Constructors

- C++ compiler automatically generates a **default copy constructor**
  - Constructor called when an object is allocated and initializes the object to be a copy of another object of the same type
  - Signature would look like `Complex(const Complex &);`
  - Called by either of the options shown in the code
    - **Simply performs an element by element copy**

- C++ compiler automatically generates a **default assignment function**
  - Called when you assign to an object that is already allocated (memory already exists)
  - **Simply performs an element by element copy**
  - `Complex& operator=(const Complex &);`

```cpp
class Complex
{
  public:
    Complex(int r, int i);
    // compiler will provide by default:
    // Complex(const Complex&);
    // Complex& operator=(const Complex&);
    ~Complex()
  private:
    int real, imag;
};

int main()
{
  Complex c1(2,3), c2(4,5)
  Complex c3(c1); // copy constructor
  Complex c4 = c1; // copy constructor
  c4 = c2; // default assignment oper.
  // c4.operator=(c2)
}
```
Assignment & Copy Constructors

- C++ compiler automatically generates a **default** copy constructor
- C++ compiler automatically generates a **default** assignment function
- See picture below of what a1 looks like as it is constructed

```cpp
class MyArray
{
public:
    MyArray(int d[], int num); //normal
    ~MyArray();
    int len; int *dat;
};

// Normal constructor
MyArray::MyArray(int d[], int num)
{
    dat = new int[num]; len = num;
    for(int i=0; i < len; i++)
    {
        dat[i] = d[i];
    }
}

int main()
{
    int vals[] = {9,3,7,5};
    MyArray a1(vals,4);
    MyArray a2(a1); // calls default copy
    MyArray a3 = a1; // calls default copy
    MyArray a4;
    a4 = a1; // calls default assignment

    // how are the contents of a2, a3, a4 related to a1

    vals  0 1 2 3
          9 3 7 5

    a1.dat 0x200 a1.len 4
    0x200 0 1 2 3

    0x200 0 1 2 3
    After 'new'

    9 3 7 5
    After constructor
```
Assignment & Copy Constructors

Default copy constructor and assignment operator make a **SHALLOW COPY** (data members only) rather than a **DEEP COPY** (data members + what they point at)

```cpp
class MyArray
{
    public:
        MyArray(int d[], int num); //normal
        ~MyArray();
        int len; int *dat;
};

// Normal constructor
MyArray::MyArray(int d[], int num)
{
    dat = new int[num]; len = num;
    for(int i=0; i < len; i++)
    {
        dat[i] = d[i];
    }
}

int main()
{
    int vals[] = {9,3,7,5};
    MyArray a1(vals,4);
    MyArray a2(a1); // calls default copy
    MyArray a3 = a1; // calls default copy
    MyArray a4;
    a4 = a1; // calls default assignment
    // how are the contents of a2, a3, a4 related to a1
}
```
When to Write Copy Constructor

• Default copy constructor and assignment operator ONLY perform SHALLOW copies
  – SHALLOW COPY (data members only)
  – DEEP copy (data members + what they point at)
  – [Like saving a webpage to your HD...it makes a shallow copy and doesn't copy the pages linked to]

• You SHOULD/MUST define your own copy constructor and assignment operator when a DEEP copy is needed
  – When you have pointer data members that point to data that should be copied when a new object is made
  – Often times if your data members are pointing to dynamically allocated data, you need a DEEP copy

• If a Shallow copy is acceptable, you do NOT need to define a copy constructor
Defining Copy Constructors

- Same name as normal constructor but should take in an argument of the object type:
  - Usually a const reference
- `MyArray(const MyArray&)`;

```cpp
class MyArray
{public:
    MyArray(int d[], int num);
    MyArray(const MyArray& rhs);
    ~MyArray();
private:
    int *dat; int len;
}
// Normal constructor
MyArray::MyArray(int d[], int num)
{
    dat = new int[num]; len = num;
    // copy values from d to dat
}
// Copy constructor
MyArray::MyArray(const MyArray &rhs)
{
    len = rhs.len; dat = new int[len];
    // copy from rhs.dat to dat
}

int main()
{
    intvals[] = {9,3,7,5};
    MyArray a1(vals,4);
    MyArray a2(a1);
    MyArray a3 = a1;
    // how are the contents of a2 and a1 related?
}
```
Implicit Calls to Copy Constructor

- Recall pass-by-value passes a copy of an object...If defined the copy constructor will automatically be called to make this copy otherwise the default copy will perform a shallow copy.

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

// Copy constructor
Complex::Complex(const Complex & c)
{
    cout << "In copy constructor" << endl;
    real = c.real; imag = c.imag;
}

// ** Copy constructor called for pass-by-value
int dummy(Complex rhs)
{
    cout << "In dummy" << endl;
}

int main()
{
    Complex c1(2,3), c2(4,5);
    int x = dummy(c1);
    // ** Copy Constructor called on c1 **
}
```
Copy Constructos

• Write a prototype for the constructor that would want to be called by the red line of code

• Now we see why the first option can't be right...because to pass c1 by value requires a call to the copy constructor which we are just now defining (circular reference/logic)
  – Complex(Complex)
    • We will see that this can't be right...

• The argument must be passed by reference
  – Complex(const Complex &)

```cpp
class Complex
{
    public:
        Complex(int r, int i);
        Complex(Complex c); // Bad b/c pass
                        // by value req. copy to be made
                        // ...chicken/egg problem
        Complex(const Complex &c); // Good
    ~Complex()
    private:
        int real, imag;
};

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

• Add a copy constructor to your Str class
Defining Copy Assignment Operator

• Operator=() is called when an object already exists and then you assign to it
  – Copy constructor called when you assign during a declaration:
    – E.g. MyArray a2=a1;
• Can define operator for '=' to indicate how to make a copy via assignment
• Gotchas?

```cpp
class MyArray
{
public:
    MyArray();
    MyArray(int d[], int num);
    MyArray(const MyArray& rhs);
    MyArray& operator=(const MyArray& rhs);
    ~MyArray();
    int* dat; int len;
}

MyArray::MyArray(const MyArray &rhs) {
    len = rhs.len; dat = new int[len];
    // copy from rhs.dat to dat
}

MyArray& MyArray::operator=(const MyArray &rhs) {
    len = rhs.len; dat = new int[len];
    // copy from rhs.dat to dat
}

int main()
{
    intvals[] = {9,3,7,5};
    MyArray a1(vals,4);
    MyArray a2;
    a2 = a1; // operator=() since a2 already exists
}
```
Defining Copy Assignment Operator

- Gotchas?
  - Dest. object may already be initialized and simply overwriting data members may lead to a memory leak
  - Self assignment (which may also lead to memory leak or lost data)

```cpp
class MyArray
{
public:
    MyArray();
    MyArray(int d[], int num);
    MyArray(const MyArray& rhs);
    MyArray& operator=(const MyArray& rhs);
    ~MyArray();
    int *dat; int len;
}

MyArray::MyArray(const MyArray &rhs){
    len = rhs.len;
    dat = new int[len];
    // copy from rhs.dat to dat
}
MyArray& MyArray::operator=(const MyArray &rhs){
    if(this == &rhs) return *this;
    if(dat) delete dat;
    len = rhs.len;
    dat = new int[len];
    // copy from rhs.dat to dat
    return *this;
}

int main()
{
    int vals1[] = {9,3,7,5}, vals2[] = {8,3,4,1};
    MyArray a1(vals1,4);
    MyArray a2(vals2,4);
    a1 = a1; a2 = a1;
}
```
Assignment Operator Practicals

- RHS should be a const reference
  - Const so we don't change it
  - Reference so we don't pass-by-value and make a copy (which would actually call a copy constructor)
- Return value should be a reference
  - Allows for chained assignments
  - Should return (*this)
  - Reference so another copy isn't made

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

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

int main()
{
    Complex c1(2,3), c2(4,5);
    Complex c3, c4;
    c4 = c3 = c2;
    // same as c4.operator=(c3.operator=(c2));
}
```
Assignment Operator Overloading

- If a different type argument can be accepted we can overload the = operator

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

Complex &Complex::operator=(const int & r)
{
    real = r; imag = 0;
    return *this;
}

int main()
{
    Complex c1(3, 5);
    Complex c2, c3, c4;
    c2 = c3 = c4 = 5;
    // c2 = (c3 = (c4 = 5));
    // c4.operator=(5); // Complex::operator=(int&)
    // c3.operator=(c4); // Complex::operator=(Complex&)
    // c2.operator=(c3); // Complex::operator=(Complex&)
    return 0;
}
```
Copy Constructor Summary

• If you are okay with a shallow copy, you don’t need to define a copy constructor or assignment operator

• **Rule of Three:**
  – Usually if you have dynamically allocated memory, you’ll need a **copy constructor**, an **assignment operator**, and a **destructor** (i.e. if you need 1 you need all 3)

• Copy constructor should accept a const reference of the same object type

• Assignment operators should be careful to cleanup initialized members and check for self-assignment

• Assignment operators should return a reference type and return *this
Exercises

• Add an assignment operator to your Str class
• Also add a '+'=' operator to your Str class