CSCI 104
Copy Semantics

Mark Redekopp
David Kempe
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

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

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

```cpp
class Complex
{
    public:
        Complex(int r, int i);

        // What constructor definition do I need for c3's declaration below

        ~Complex()
        private:
            int real, imag;
};

int main()
{
    Complex c1(2,3), c2(4,5)
    Complex c3(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 &);`
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
```
Assignment & Copy Constructors

Initial values of \( d[] \):

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
9 & 3 & 7 & 5 \\
\end{array}
\]

```
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 Constructors

- 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 &)
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.
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