

# CSCI 104

## Copy Semantics

CSCI 104 Teaching Team  
Spring 2025

# Review from CS 103 [1]

- What is the correct prototype for the copy constructor call when c3 is created in the code to the right?
  - **Complex(Complex);**
  - Complex(Complex &)
  - **Complex(const Complex &)**

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

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

private:
    double real, imag;
};

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

# Review from CS 103 [2]

## Which function?

- For each of the following, identify whether the **copy constructor** is called or the **assignment operator**
  - **Complex c1;**  
**Complex c2 = c1;**
  - **Complex c1;**  
**Complex c2(c1);**
  - **Complex c1, c2;**  
**c2 = c1;**

## Default Versions

- What kind of copy does the default copy constructor and assignment operator perform?

```
class MyArray
{
    ...
private:
    int* data; // ptr to dynamic array
    size_t len;
};
```

# Review from CS 103 [3]

## State the Rule of 3

- The rule of 3:

## Assignment Operator Specifics?

- What extra considerations does the assignment operator need to handle vs. the copy constructor?
- What should operator= return?

```
class MyArray
{

private:
    int* data; // ptr to dynamic array
};

MyArray& operator=(const MyArray& other)
{

}
```

Copy constructors and assignment operators

# **COPY SEMANTICS**

# 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

0x7e0

cards[52]	37	21	4	9	16	43	20	39
top_index	0							

d2

0x2a0

cards[52]	41	27	8	39	25	4	11	17
top_index	1							

d1

d1 is implicitly passed to shuffle()

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

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

poker.cpp

```
#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;
    }
}
```

Actual code you write

this

0x2a0

```
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;
    }
}
```

deck.cpp

Compiler-generated code

# Another Use of 'this'

- This can be used to resolve scoping issues with similar named variables
  - Exercise: `this_scope`

```
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

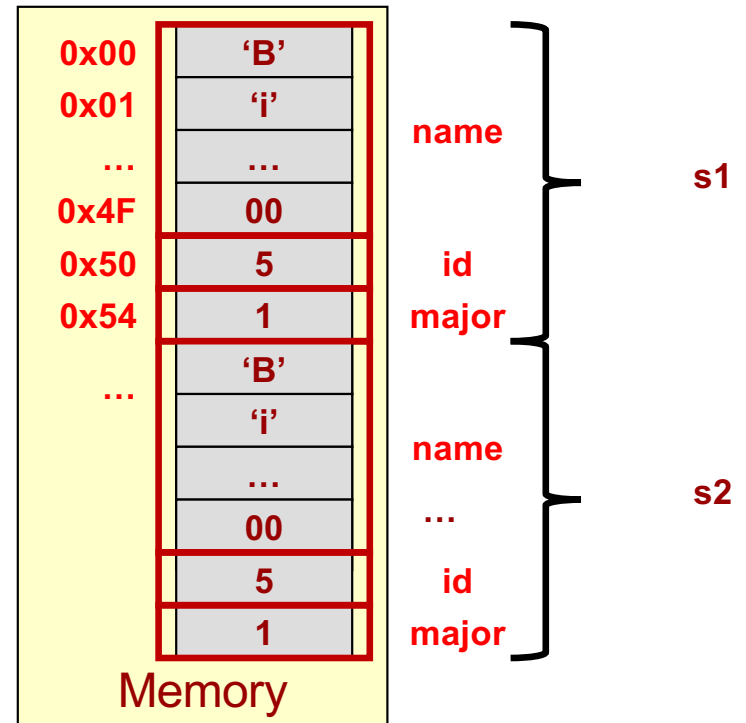
- Assigning one struct or class object to another will perform an element by element copy of the source struct/class to the destination struct/class

```
#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

```
#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
}
```

```
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;
};
```

```
Student::Student()
{
    _name = "", _id = 0; _gpa = 2.0;
}
Student::Student(string name, int id, double gpa)
{
    _name = name; _id = id; _gpa = gpa;
}
```

Student.h

Student.cpp

# Copy Constructors

- Write a prototype for the constructor that would want to be called by the red line of code
- Realm of Reasonable Answers:
- We want a constructor that will build a new Complex object (c3) by making a copy of another (c1)

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

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

private:
    double real, imag;
};

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

# 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 &)
    - Possible
  - Complex(const Complex &)
    - Best! (Making a copy shouldn't change the input argument, thus 'const')
- We want a constructor that will build a new Complex object (c3) by making a copy of another (c1)

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

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

private:
    double 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 &);**

```

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

**Class Complex**

int real\_  
int imag\_

**c4**

int real\_  
int imag\_

**c2**

int real\_  
int imag\_

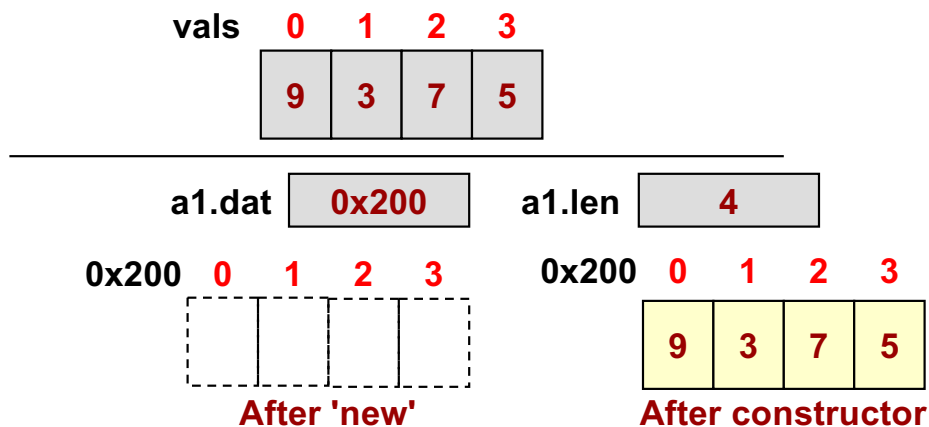
# Assignment & Copy Constructors

- 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

```

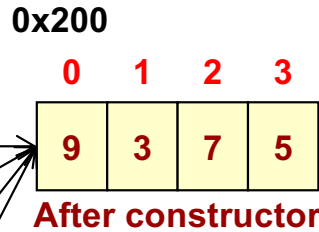
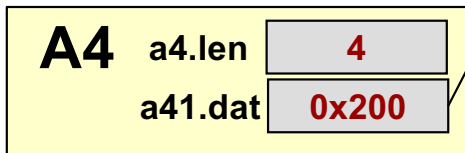
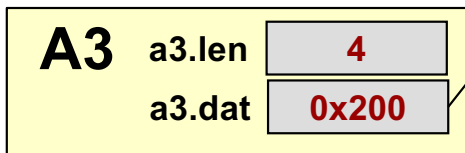
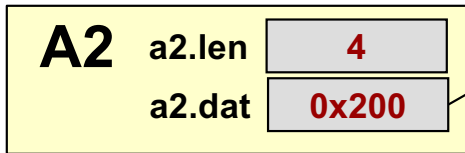
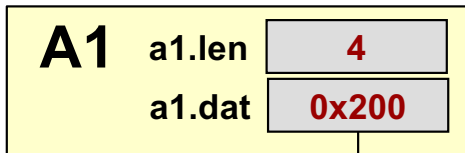
class MyArray
{
public:
    MyArray(int d[], int num); //normal
    ~MyArray();
    int len; int *dat;
};
// Normal constructor
MyArray::MyArray(int d[], int num)
{
    dat = new int[num]; len = num;
    for(int i=0; i < len; i++){
        dat[i] = d[i];
    }
}

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

vals	0	1	2	3
	9	3	7	5



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

```
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&);**

```
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

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

```
class Complex
{
public:
    Complex();
    Complex(double r, double 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:
    double real, imag;
};

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

# 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?**

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

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)

```
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

```
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

```
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 For Home

- Suppose you are given a class that implements a singly-linked of integers (with a head pointer data member)
- Write a '-=' operator that takes one element and removes it from the list if it exists
- Write a '==' operator that checks whether the contents and order of one list matches another

```
#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!";
    }

    m2.push_back(7);
    m2 -= 5; // now m2 would just have [7]

    if(m1 == m2){
        cout << "Should not print!"; << endl;
    }
    return 0;
}
```



# SOLUTIONS

# Review from CS 103 [1]

- What is the correct prototype for the copy constructor call when c3 is created in the code to the right?
  - **Complex(Complex);**
    - We will see that this can't be right...
  - **Complex(Complex &)**
    - Possible
  - **Complex(const Complex &)**
    - Best! (Making a copy shouldn't change the input argument, thus 'const')

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

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

private:
    double real, imag;
};

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

# Review from CS 103 [2]

## Which function?

- For each of the following, identify whether the **copy constructor** is called or the **assignment operator**
  - **Complex c1;**  
**Complex c2 = c1;**
    - **Copy constructor**
  - **Complex c1;**  
**Complex c2(c1);**
    - **Copy constructor**
  - **Complex c1, c2;**  
**c2 = c1;**
    - **Assignment operator**

## Default Versions

- What kind of copy does the default copy constructor and assignment operator perform?
  - **Shallow copy (member by member copy)**

```
class MyArray
{
    ...
private:
    int* data; // ptr to dynamic array
    size_t len;
};
```

# Review from CS 103 [3]

## State the Rule of 3

- The rule of 3:
  - If a class needs a user-defined version of any one of the 3: copy constructor, assignment operator, or destructor, it needs ALL 3.

```
class MyArray
{

private:
    int* data; // ptr to dynamic array
};

MyArray& operator=(const MyArray& other)
{

}

}
```

## Assignment Operator Specifics?

- What extra considerations does the assignment operator need to handle vs. the copy constructor?
  - Must clean up old resources before copying
  - Beware of self assignment
- What should operator= return?
  - A reference to an instance of the class which should be \*this;