EECS 280 – Lecture 15

Deep Copies and the Big Three

3/9/2022

Destructors and Polymorphism

```
class Player {
public:
  ...
  ~Player() {}
  ...
};
```

Because the destructor is non-virtual, we end up using this one!

```
class SimplePlayer : public Player {
  vector<Card> hand;
public:
  ...
  ~SimplePlayer() {}
  ...
};
```

The hand vector does not get destructed.

```
int main() {
  Player *player = new SimplePlayer(...);
  ... // do stuff with player
  delete player;
}
```

This will break.

Static type here is `Player`.

Destructors and Polymorphism

```
class Player {
public:
  ...
  virtual ~Player() {}
  ...
};
```

```
class SimplePlayer : public Player {
  vector<Card> hand;
public:
  ...
 virtual ~SimplePlayer() {}
  ...
};
```

If there are subtypes, ALWAYS make the destructor virtual.

Now use this one! It implicitly calls the destructor for hand.

```
int main() {
  Player *player = new SimplePlayer(...);
  ... // do stuff with player
  delete player;
}
```

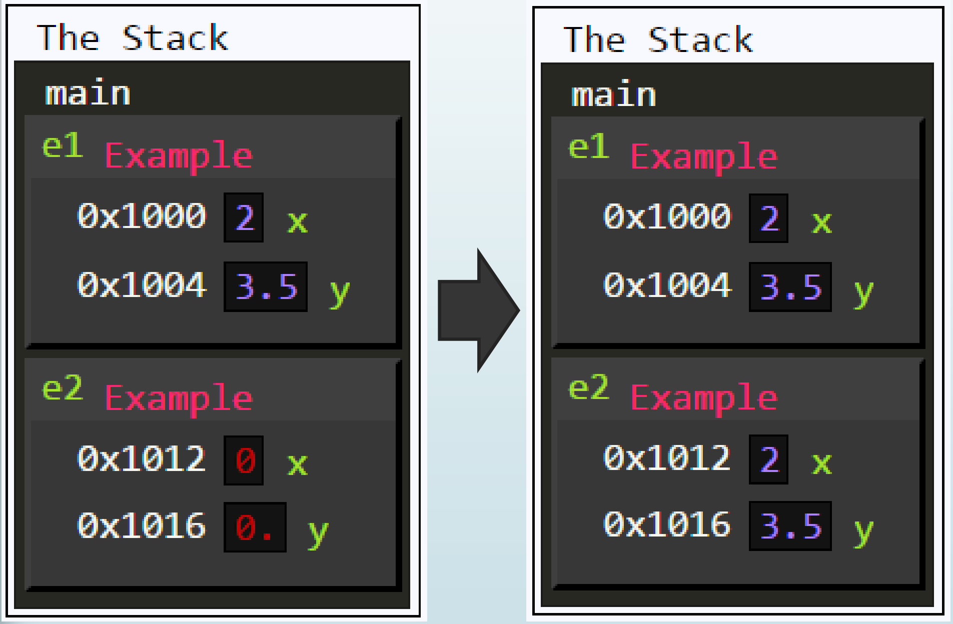
This will work.

Dynamic type here is `SimplePlayer`.

Recall: Copying Compound Objects

- The built-in behavior for copying compound objects is just a straightforward member-by-member copy.

```
class Example {
private:
  int x; double y;
public:
  Example(int x_in,
          double y_in)
   : x(x_in), y(y_in) { }
};

int main() {
  Example e1(2, 3.5);

  // init e2 as copy of e1
  Example e2(e1);

  // This syntax is equivalent
  // Example e2 = e1;
}
```



But how does this work internally?

Two Kinds of Copies

- Two contexts in which we can copy the value from one object to another:
  - **Initialization (also used for parameter passing)**

<u>Initialization</u> is giving a first-time value as part of creating the object.

```
int func(Example a); // pass by value
int main() {
  Example a1(2, 3.5);
  Example a2(a1);  // init a2 as copy of a1
  Example a3 = a1; // init a3 as copy of a1
  func(a1);        // init parameter a as copy of a1
}
```

The Copy Constructor

- When a compound object is **initialized** from another of the same type, a **copy constructor** is used.

```
class Example {
private:
  int x; double y;
};

int main() {
  Example e1(2, 3.5);

  // init e2 as copy of e1
  Example e2(e1);
}
```

Question

Which of these is a valid copy ctor for Example?

```
class Example {
public:

A   Example(int x_in, double y_in)
     : x(x_in), y(y_in) { }

B   Example(const Example &other)
     : x(other.x), y(other.y) { }

C   Example(const Example &other)
     : Example(other) { }

D   Example(Example other)
     : x(other.x), y(other.y) { }

E   Example(const Example &other) {
      x = other.x;
      y = other.y;
    }
};
```

The Copy Constructor

- Why is the parameter for the copy constructor passed by reference?

```
class Example {
public:
  Example(const Example &other);
};
```

YES

- What would happen if we used pass-by-value instead?

```
class Example {
public:
  Example(const Example other);
};
```

NO

  - Passing an `Example` by value uses the copy constructor…this leads to infinite recursion!

Implicitly Defined Copy Constructor

- In the previous exercise, we wrote out a copy constructor for the `Example` class:

```
class Example {
public:
  Example(const Example &other)
    : x(other.x), y(other.y) { }
};
```

- **HOWEVER**, all `classes`[1] **already** have a built-in copy constructor just like this that the compiler provides for you.
  - It just does a **member-by-member** copy.
  - The built-in version is provided even if you made other constructors (unlike the built-in default ctor).

1 And `structs`, technically.

UnsortedSet Copy Constructor

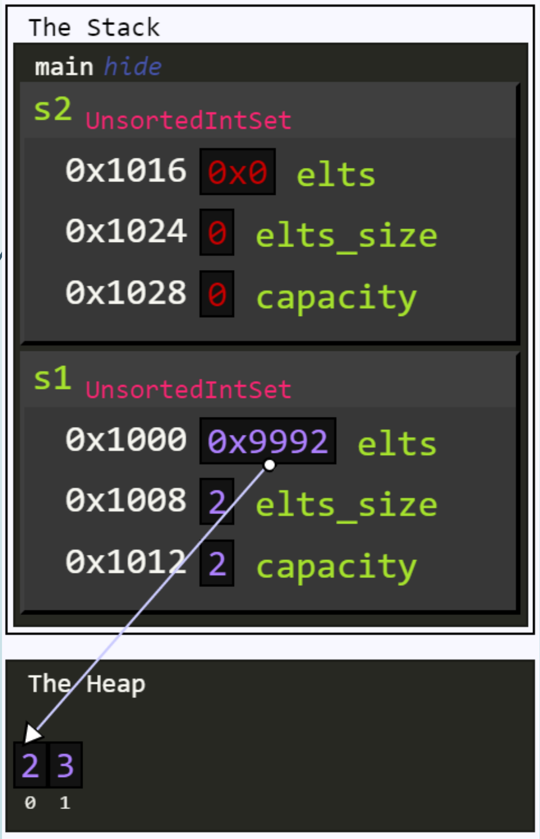
- Here's the built-in copy constructor for `UnsortedSet`:

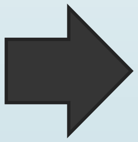
```
template <typename T>
class UnsortedSet {
private:
  T *elts;
  int capacity;
  int elts_size;
public:
  UnsortedSet(const UnsortedSet &other)
    : elts(other.elts),
      capacity(other.capacity),
      elts_size(other.elts_size) { }
};
```

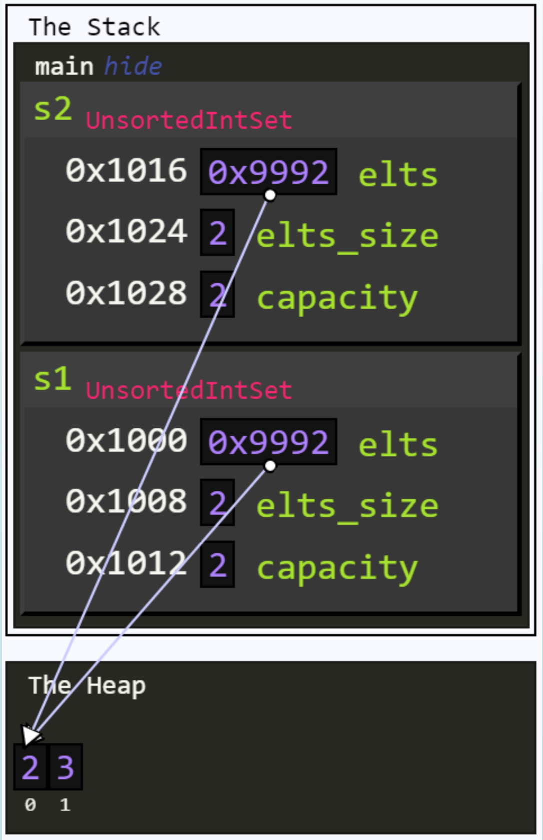
- Is this the copying behavior we would want for `UnsortedSet`?

The Shallow Copy Problem

- The built-in **shallow** copying behavior doesn't work well with objects used **indirectly**.
  - It doesn't follow pointers!
  - It just copies the address.







Shallow Copies vs. Deep Copies

Exercise: Deep Copy

- Write a custom version of the `UnsortedSet` copy constructor that implements a **deep copy**.

```
template <typename T>
class UnsortedSet {
public:
  UnsortedSet(const UnsortedSet &other)



                                 {





  }
};
```

Solution: Custom Copy Constructor

- We need a **deep copy** for `UnsortedSet`.

```
template <typename T>
class UnsortedSet {
public:
  UnsortedSet(const UnsortedSet &other)
    : elts(new T[other.capacity]),
      capacity(other.capacity),
      elts_size(other.elts_size) {

    for (int i = 0; i < elts_size; ++i) {
      elts[i] = other.elts[i];
    }

  }
};
```

Allocate our own array with same size as the other.

Copy over each element individually.

It's ok to access `other.elts`, even though it's private. We're still in the `UnsortedSet` scope!

Break Time!

Pooja Sankar

Founder and CEO of Piazza

Break Time!

Guido van Rossum

Creator of the Python programming language

Break Time!

We'll start again in one minute.

Two Kinds of Copies

- Two contexts in which we can copy the value from one object to another:
  - Initialization (also used for parameters)

```
int func(Example a); // pass by value
int main() {
  Example a1(2, 3.5);
  Example a2(a1);  // init a2 as copy of a1
  Example a3 = a1; // init a3 as copy of a1
  func(a1);        // init parameter a as copy of a1
}
```

  - **Assignment**

```
int main() {
  Example a1(2, 3.5);
  Example a2(7, 3.14);
  a2 = a1; // set a2 to be a copy of a1
}
```

<u>Initialization</u> is giving a value as part of creating the object.

<u>Assignment</u> is giving a new value to an object that already exists.

The Assignment Operator

- **Assigning** the value of one compound object to another uses an overloaded **assignment operator**.

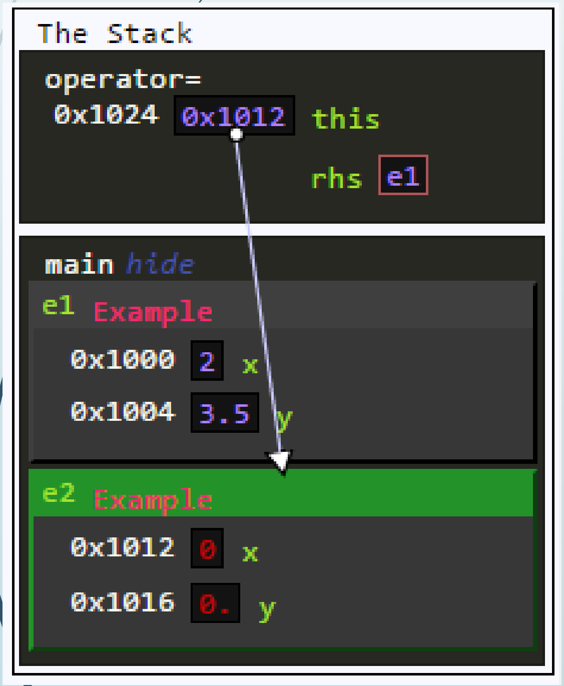
```
class Example {
public:
  Example & operator=(const Example &rhs);
};
```

- If you don't provide your own assignment operator, the compiler implicitly makes one for you.
  - It just does a **member-by-member** copy.
  - For `Example`, it would look something like this…

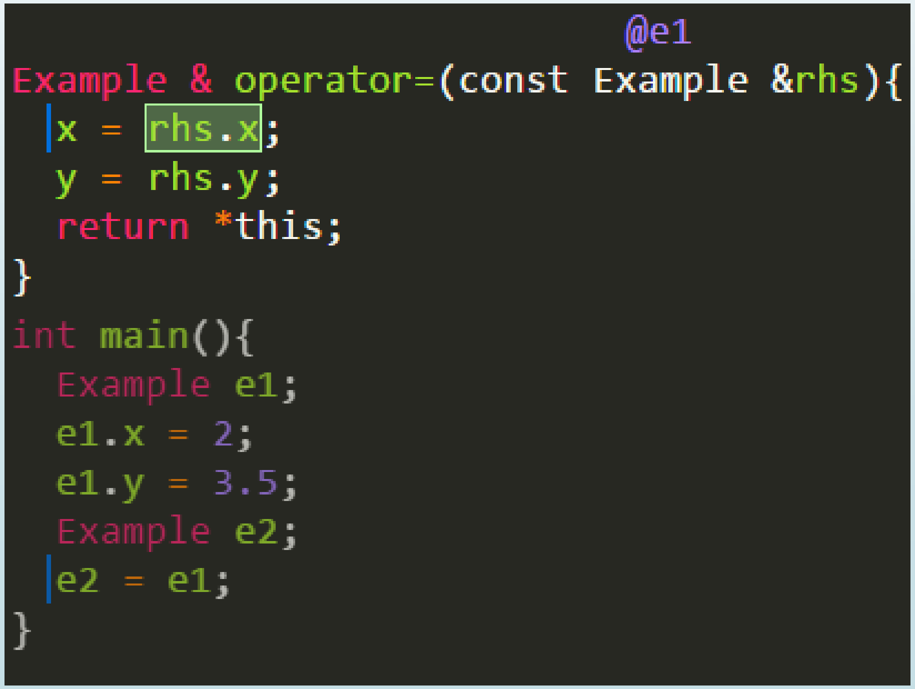
```
class Example {
public:
  Example & operator=(const Example &rhs) {
    x = rhs.x;
    y = rhs.y;
    return *this;
  }
};
```

The Assignment Operator

- Overloads for the assignment operator must be implemented as **member functions**.
- When the function is called…
  - The **this pointer** points to the **left hand side**.
  - The **parameter** is a **reference** to the **right hand side**.



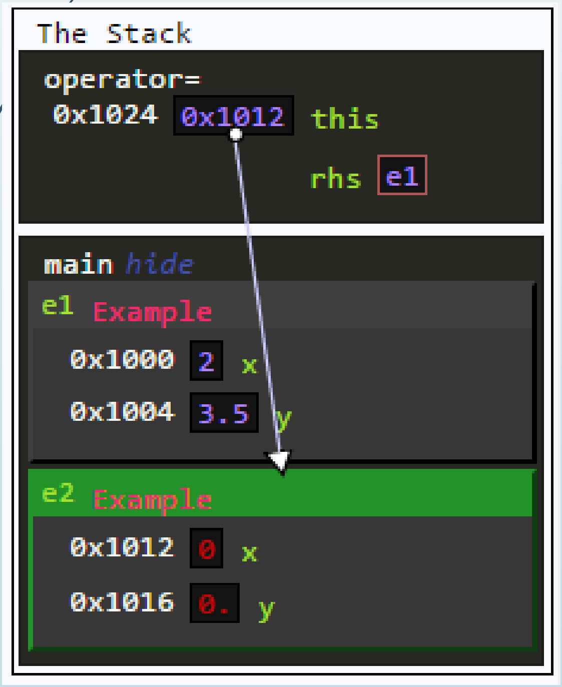
```
The Stack
operator=
 0x1024  0x1012  this
                 rhs  e1

main hide
e1 Example
  0x1000  2    x
  0x1004  3.5  y
e2 Example
  0x1012  0    x
  0x1016  0.   y
```



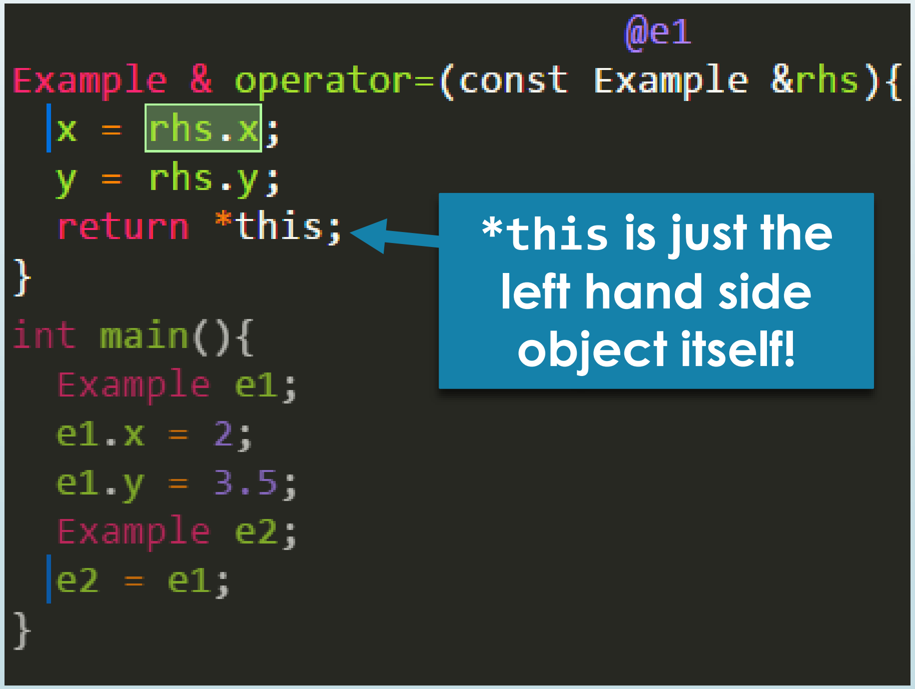
```
                                   @e1
Example & operator=(const Example &rhs){
  x = rhs.x;
  y = rhs.y;
  return *this;
}
int main(){
  Example e1;
  e1.x = 2;
  e1.y = 3.5;
  Example e2;
  e2 = e1;
}
```

return *this

- Semantically, an assignment expression is supposed to evaluate back into its left hand side (the object assigned to).
- Example: `cout << (x = 3) << endl;`
- We first assign 3 into x, but then the whole thing in parentheses turns back into x, which is then printed.



```
The Stack
operator=
 0x1024 0x1012 this
             rhs e1
main hide
e1 Example
  0x1000 2   x
  0x1004 3.5 y
e2 Example
  0x1012 0   x
  0x1016 0.  y
```



```
                              @e1
Example & operator=(const Example &rhs){
  x = rhs.x;
  y = rhs.y;
  return *this;
}
int main(){
  Example e1;
  e1.x = 2;
  e1.y = 3.5;
  Example e2;
  e2 = e1;
}
```

***this is just the left hand side object itself!**

Copy Ctor vs. Assignment Operator

Custom Assignment Operator

- We need a **deep copy** for `UnsortedSet`.

```
template <typename T>
class UnsortedSet {
public:
  UnsortedSet & operator=(const UnsortedSet &rhs) {
    if (this == &rhs) { return *this; }

    delete[] elts;
    elts = new T[rhs.capacity];
    capacity = rhs.capacity;
    elts_size = rhs.elts_size;

    for (int i = 0; i < elts_size; ++i) {
      elts[i] = rhs.elts[i];
    }

    return *this;
  }
};
```

Check for self-assignment.

Kill old array.

Make new array.

Copy over all the elements.

Return the LHS object.

Recap: Deep Copies

- Sometimes we need to provide custom implementations for the **copy constructor** and **assignment operator**.
  - The built-in behavior gives us a **shallow copy**, and that is very bad.
- `UnsortedSet` is an example of this.
  - In this case, a proper copy should be **deep**.
  - A copy should have its own dynamically allocated array – not just a pointer to someone else's!

The Big Three

3

The Rule of The Big Three

- The Big Three
  - Destructor
  - Copy Constructor
  - Assignment Operator
- The rule of the big three is:
  - If you need to provide a **custom** implementation for **any** of them…

    …you almost certainly need to provide a custom implementation for **all** of them.

Implicitly Defined Big Three

- All objects have the big three.
  - If you don't provide a custom version, the compiler provides **implicitly defined** versions for you.
- Implicit destructor
  - No special cleanup
- Implicit copy constructor
  - Shallow copy
- Implicit assignment operator
  - Shallow copy

Custom Big Three

- When do we need our own **custom** versions?
  - If you need a deep copy.
  - You need a deep copy if the object owns and manages any resources (e.g. dynamic memory).
- Hints:
  - Check the constructor. If it creates dynamic memory, you probably need the big three.
  - Look at the members. If some of them are pointers, you might need the big three.

The Big Three

- Destructor
  1. Free resources[1]
- Copy Constructor
  1. Copy regular members from `other`
  2. Deep copy resources from `other`
- Assignment Operator
  1. Check for self-assignment
  2. Free old resources
  3. Copy regular members from `rhs`
  4. Deep copy resources from `rhs`
  5. return `*this`

1 The "resource" we often see in 280 is dynamic memory.

Exercise: Compound Copies

```
class MyClass {
private:
  string s;
public:
  MyClass(const string &s_in) : s(s_in) {
    cout << "MyClass ctor " << s << endl;
  }

  MyClass(const MyClass &other) : s(other.s) {
    cout << "MyClass copy ctor " << s << endl;
  }

  MyClass &operator=(const MyClass &rhs){
    cout << "MyClass assign " << s
         << " to be " << rhs.s << endl;
    s = rhs.s;
    return *this;
  }

  ~MyClass() {
    cout << "MyClass dtor " << s << endl;
  }
};
```

- What will this code print?

```
void func(MyClass &x,
            MyClass y) {
  MyClass z = x;
}

int main() {
  MyClass a("apple");
  MyClass b("banana");
  MyClass c("craisin");

  func(a, b);

  // Careful!
  MyClass c2 = c;
  c2 = c;
}
```

See file `L16.1_classes` on Lobster.

Solution: Compound Copies

- What will the following code print?

```
MyClass ctor apple
MyClass ctor banana
MyClass ctor craisin
MyClass copy ctor banana
MyClass copy ctor apple
MyClass dtor apple
MyClass dtor banana
MyClass copy ctor craisin
MyClass assign craisin to be craisin
MyClass dtor craisin
MyClass dtor craisin
MyClass dtor banana
MyClass dtor apple
```

```
void func(MyClass &x,
          MyClass y) {
  MyClass z = x;
}

int main() {
  MyClass a("apple");
  MyClass b("banana");
  MyClass c("craisin");

  func(a, b);

  // Careful!
  MyClass c2 = c;
  c2 = c;
}
```