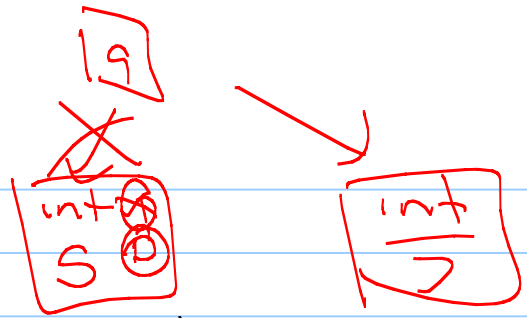
CS180 - Destructors

Note Title 9/7/2012

Announcements

- HW2 - due Friday
- Email 24 hours in advance
- Door code: 80386

Garbage Collection



In Python, variables that are no longer in use are automatically destroyed.

Pros: easy!

Cons: slow

C++

In C++, things are sometimes handled for you.

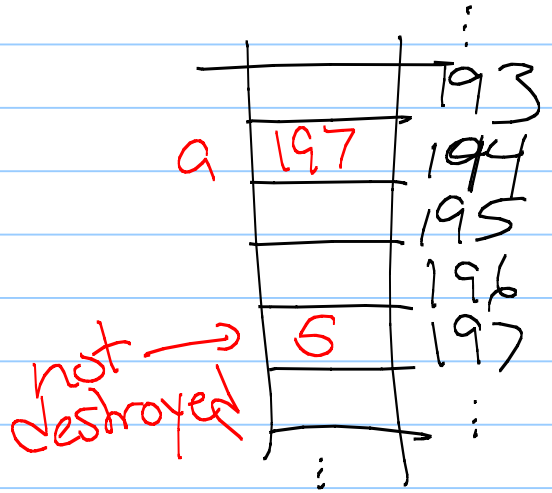
Basically, any standard variable is automatically destroyed at the end of its scope.

This holds for <u>any</u> type of variable!

Problem: Pointers

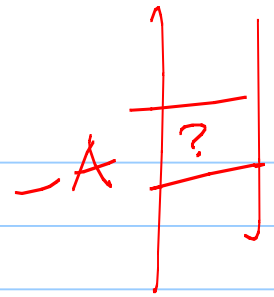
While the pointer variable is deleted, the spot you created with "new" is not.

```
int main() {
    int * a = new int(5);

    delete a;
} // a is destroyed
```



Rule: If you have a new, must have a delete!

```
class MyFloatVec {
private:
    int _size; // size of this array
    float *_A; // panter to my array

public:
    MyFloatVec ( int s=10 ) :_size(s) {
        _A = new float[_size];
    }
```



Copy Constructor

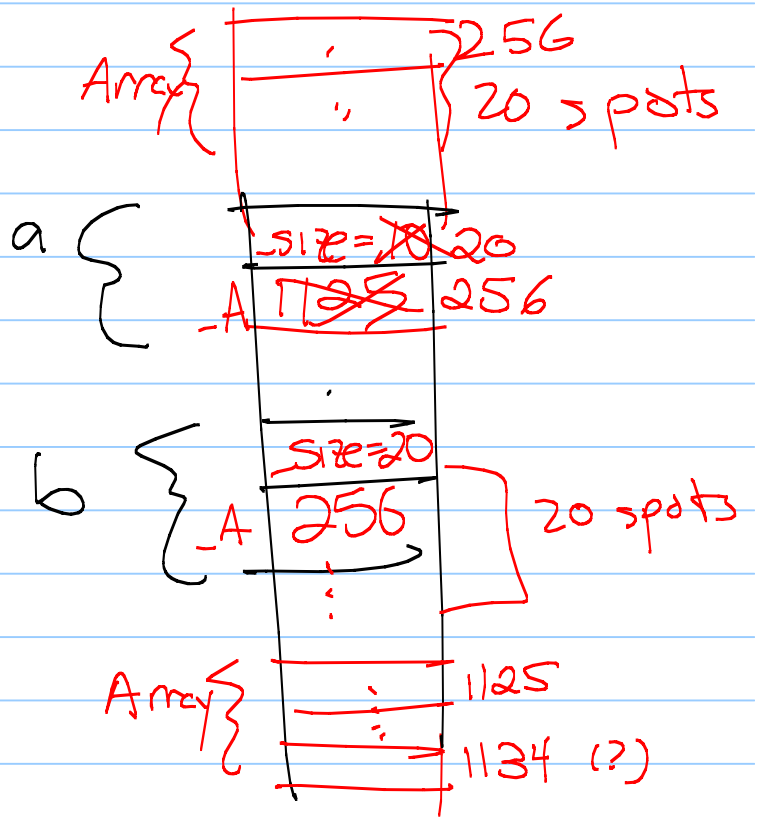
Consider that MyFloatVec class.

What if we have 2 + set a = b?

or MyFloatVec a(b);

a._size = b._size

a._A = b._A



Problems

① shallow copy

② Didn't clear old data

To avoid shallow copies we need to make a copy constructor function.

```
MyFloatVec(const MyFloatVec & other) {
    _size = other._size;
    _A = new float[_size];
    for (int i=0; i<_size; i++)
        _A[i] = other._A[i];

}
```

Another issue:

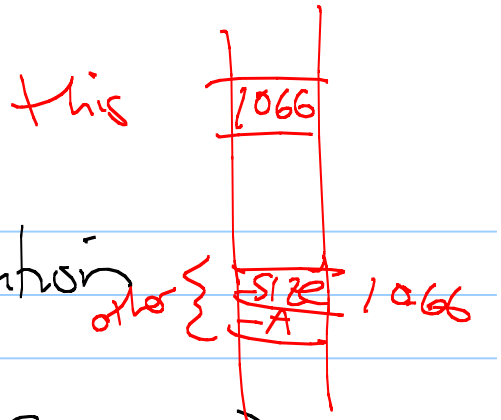
```
MyFloatVec c;
c = a;
```

What does this do?

Shallow copy

± didn't deallocate old data

a = a



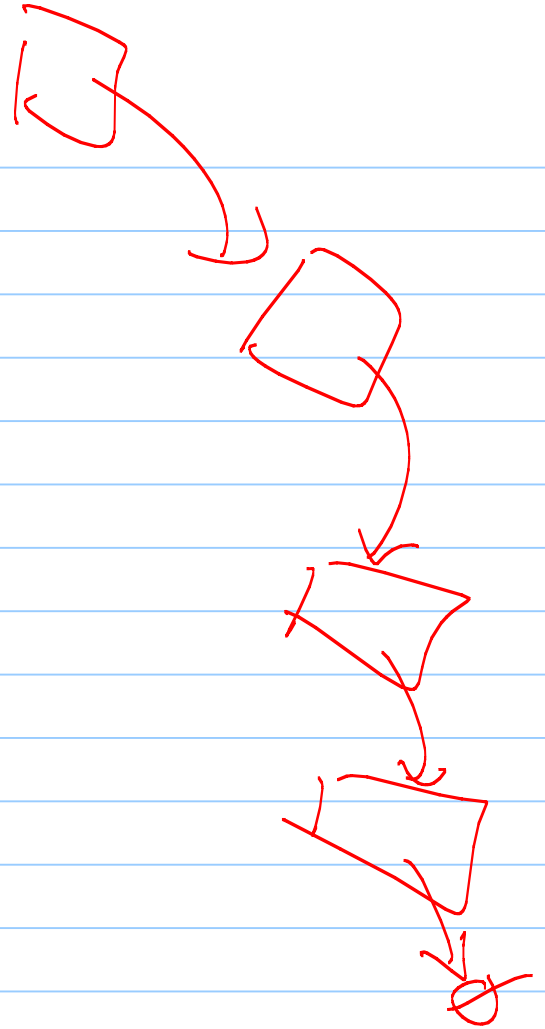
Solution: rewrite the "=" operation

```
MyFloatVec Operator=(const MyFloatVec& other) {
    if (this != &other) {
        // copy data over + deallocate
        _size = other._size;
        float* temp = new float[_size];
        for (int i=0; i<_size; i++)
            temp[i] = other._A[i];
        delete _A;
        _A = temp;
    }
    return *this;
}
```

Housekeeping Functions

1. Copy Constructor
2. Operator=
3. Destructor:

```
~MyFloatVect() {
    delete _A;
}
```



Enum: user defined types

```
enum Color {RED, BLUE, GREEN};
           0     1     2

Color sky = BLUE;
Color grass = GREEN;
Color fire = 0;   <- or
                  1
if (sky == BLUE)
    cout << "It's nice out today!" << endl;
```

Structs

useful for simple collections of objects

Ex: `enum MealType {NO_PREF, VEG, REGULAR, KOSHER};`

```
struct Passenger {
    string name;
    MealType mealPref;
    bool isFreqFlyer;
    string freqFlyerNo;
}
```

Using structs

We can then create instances of a struct in the program:

```
Passenger pass = { "John Smith", VEG, true,
                   "1234" };

pass.mealPref = KOSHER;
```

↑ no private data

More Complex

```
Passenger* p;
p = new Passenger;
p->name = "Barbara Wright";
p->mealPref = REGULAR;
(*p).isFreqFlyer = false;
(*p).freqFlyerNo = "None";
```

Templates

If we want a function to work for multiple classes – eg int and floats – we can template the variable type.

Ex:

```
template <typename T>          ← ItemType
T min(T a, T b) {
    if (a<b)
        return a;
    else
        return b;
}
```

Important:
Will work for any class with appropriate operators!

Ex:
```
int x = 53;
int y(96);

int z = min(x,y);
```

Works for any class with < operator!

```
string a = "Hello";
string b = "Goodbye";

cout << min(a,b) << endl;
```

Templates in classes

These work in classes, also.

Important in data structures, so our code will make a list of ints or strings or lists!

Using a template:

```
MyList<int>    list1;

MyList<string> list2;
list1.append(3);
list2.append("Hello");
list1.append(6.2); <-- error
```