CS 180 - Destructors

Announcements

- HW 2 - due Friday
- Email 24 hours in advance
- Pool code : 80386
Garbage Collection

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

Pros: easy!

Cons: slow
In C++, things are sometimes handled for you.

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

This holds for any type of variable!
Problem: Pointers

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

```c
int main() {
    int *a = new int(5);  // 193
    delete a;
    // 194
    // 195
    // 196
    // 197
    // 198
}
```

Rule: If you have a `new`, must have a `delete`.
class MyIntVec

private:
    int size; // size of this array
    float *A; // pointer to my array

public:
    MyIntVec (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[other._size];
  for (int i = 0; i < other._size; i++)
    A[i] = other.A[i];
}

}
Another issue:

\[
\text{MyFloatVec } c;
\]

What does this do?

Shallow copy old

\[+ \text{ didn't deallocate data}\]
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 :
   ~ My Float Vec( )
   delete -A : j
Enum: user defined types

enum Color {RED, BLUE, GREEN};

Color sky = BLUE;

Color grass = GREEN;
Color fire = 90; // ok

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 isFrogFlyer;
    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 - e.g. int and floats - we can template the variable type.

Ex:

```cpp
template <typename T> 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);

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:

```cpp
MyList <int> list1;

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