Memory Management

Two sessions ago, we began exploring classes. We started by talking about constructors, and then delved into operator overloading. We are going to touch on a few of special methods we haven't yet - the copy constructor, the assignment operator, and the destructor. Each of these plays a crucial role in memory management. But that's getting ahead of ourselves. Let's look at them in turn.

More Class Methods

Destructor

When an object is about to get deleted, C++ will call a special method called a destructor. Object destruction happens automatically when said object goes out of scope, in the case of a stack defined variable, or when delete is called, in the case of a heap allocated variable. Either way, the destructor gets called, giving us an oportunity to perform any cleanup necessary.

In C++, you define a destructor by prefixing the class name with a tilda. So for example:

```
class Person {
publc:
    // constructor
    Person();
    // destructor
    ~Person();
};
```

By the way, Python also provides a destructor. It goes by the dunder method (`__del__`) and gets called at approximately the same time.

Copy Constructor

The copy constructor is a special form of constructor which takes a const reference to an object of the class in question, and returns a new instance of that class. The copy constructor is engaged when you initialize a new instance variable from an existing instance variable. For example:

```
Person p("Frank", "Ford");
Person p2 = p1;
```

What is really going on here? A bit of syntactic sugar, to be frank. Here is the translation:

```
Person p("Frank", "Ford");
Person p2(p);
```

(and by the way, this also calls the copy constructor.)

The copy constructor's signature looks like this. (commit it to memory)

```
Person(const Person& p);
```

It takes a const reference to a variable of the same type. Its job is to initialize the new variable with the values of the old variable. If you do not create a copy constructor, one will be provided for you by the compiler. But, and this is a big but, it will often not have the intended behavior. Especially when you start getting into managing memory yourself. At this point, with the person class, there is really no need for a custom copy constructor, but here is how you would implement it:

```
Person(const Person& p) : first_name(p.first_name), last_name(p.last_name) {};
```

Assignment Operator

The assignment operator is the kissing cousin of the Copy Constructor. It's job is to make a copy of the a variable and assign it to an existing variable of the same type. Catch that? its a subtle distinction. In the copy constructor's case, we are initializing a new variable. In the assignment operator's case, we are assigning the value of an existing variable to another existing variable. Because the variable on the left hand side of the equation already exists, in cases where we have pointer variables, with owned memory, we may need to dispose of existing memory before allocating new memory and copying values. This probably makes no sense at this point, because we have not gotten into dynamic memory allocaiton. So just take it on faith that there is a reason for all of this. Anyway, the assignment operator takes the following form:

```
Person& operator=(const Person& p) {
    if (this != &p) {
        //copy
        first_name = p.first_name;
        last_name = p.last_name;
    }
    return *this;
}
```

A couple of interesting notes. We dereference the **this** pointer and return a reference to the value (combination of the signature of the return type(Person&) and the dereferece (*this)).

Right about now, you are probably wondering what the point is for these methods. They don't really seem to have any practical use. Well hold on there, because

we are about to delve into memory management, and their role in C++ will presently become clear.

So yes, next, we are going to get on with one of the most important aspects of C++ - memory management. First, we are going to talk about traditional memory management. Then we are going to delve into c+11's helper classes which make the topic much simpler.

Manual Memory Management

Managing memory manually is pretty simple; it doesnt require complex maths or twisted logic. However, it does require very rigid book keeping and forethought. As Python programmers you have been freed of the drudgery of thinking about all of this. Well, not any more. Welcome to the exciting world of managing memory.

In order to learn how to manage memory, you really only need to learn about a pair of commands - *new* an *delete*. As you might expect, new allocates memory on the heap, and delete frees memory on the heap. Here is a simple example of memory creation and deletion:

```
#include <iostream>
#include <string>

int main(){
    // allocate a string on the heap, assigning it to a pointer
    std::string *name = new std::string("Fred Flinstone");
    // use the pointer as we have already learned to do
    std::cout<< "his name is "<< *name << std::endl;
    // delete the memory
    delete name;
    return 0;
}
```

Simple eh? There is actually one variation of this when allocating an array of something. It looks like this:

```
#include <iostream<
#include <string>

int main() {
    std::string* names = new std::string[2] { std::string("fred"),  std::string("barney")};
    std::string << names[0] << " " << names[1] << std::endl;

    delete[] names;
}
```

So what is the big deal? Well let's combine this with a class and see. We are going to create a very simple Person class. Here it is:

```
class Person{
    std::string* firstname;
    std::string* lastname;
 Public:
    Person(const std::string& fn, const std::string& ln) :
    firstname(new std::string(fn)),
    lastname(new std::string(ln))
    {};

    void greet() const {std::cout << "hi my name is " << firstname << " " << lastname << std

}
```

And we will use Person in main.

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

int main() {
    Person person("Troy" "Mclure");
    person.greet();
}
```

First Problem - who is cleaning up person?

The first problem we have here is that Person allocated two strings - firstname and lastname - but didn't dealocate them. We are going to take care of that right now. To handle this automagically, we are going to implement a destructor. The destructor signature looks exactly like a default constructor, but with a tilda (~) in front. For person, that is `~Person`.

Destructor Implementation

Our Person destructor implementation is going to look like this:

```
~Person() {
    std::cout << "Person destructor called " << std::endl;

    // delete any memory we have allocated
    delete firstname;
    delete lastname;
}
```

Great. Now that we have a destructor cleaning up for us, everything is peachy. We don't have to worry about cleaning up. Right? Let's give it a try:

main.cpp

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

using std;;cout;
using std::cin;

void use_person() {
    Person bob("Bob", "Bitchen");
    bob.greet();
}

int main() {
    use_person();
    cout << "we are done"<< endl;
    return 0;
}
```

Well that worked. We are done. Memory management isn't so bad. What's all the fuss? Well hold on there bub. We aren't done. Let's take a look at some problems...

main2.cpp

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

using std;;cout;
using std::cin;

void use_person() {
    Person bob("Bob", "Bitchen");
    Person bob2 = bob;
    bob.greet();
    bob2.greet();
}

int main() {
    use_person();
    cout << "we are done"<< endl;
    return 0;
}
```

Ouch. So what is going on here? when we assign bob to bob2, the compiler generates a default copy constructor for us, and that default copy constructor

does a piecewise copy of all the variables. In other words. bob2.firstname, a pointer to string, is getting set to the *address* of bob.firstname. And when bob and bob2 go out of scope at the end of *use_person()*, they both try and delete the same memory for firstname and lastname!

So how do we fix this? Rememeber the copy constructor? This is what it is for.

The Rule of 3

The rule of 3 states that whenever you have to create a destructor to clean up memory, you have to implement a copy constructor and an assignment operator as well. So lets go ahead and do that in Person.

Person.h

```
Person {
...
public:
...
    Person(const Person& other);
    Person& operator=(const Person& other);
}
```

Person.cpp

```
#include <string>
using std::string;

...

Person::Person(const Person& other) {
    firstname = string(*other.firstname);
    lastname = string(*other.lastname);
}

// assignment operator
Person& Person::operator=(const Person& other) {
    // if the address of me is not the same as the address of other
    if(this != &other) {
        if (firstname != nullptr)
            delete firstname;
        if (lastname != nullptr)
            delete lastname;
        firstname = string(*other.firstname);
        lastname = string(*other.lastname);
    }
```

```
    return *this;
}
```

Great *now* everything should work better. Let's try our program with this better version of Person. . . .

By the way, I want you to notice one thing above. When we dereference firstname and lastname, we don't use arrow notation. This trips up people, but the reason is that "other" is not a pointer; "firstname" and "lastname" are.

Well, that didn't seem too bad really. And we are finally done right? Not so fast. I want to point out a couple of things which make may trip you up, even following the **rule of three**.

This was relatively simple, since we allocated and deallocated the memory. But there are plenty of cases where you can have a pointer to a resource which you dont own, and it can get deleted, leaving you with a dangling pointer.

Fortunately, there is a simpler way of doing this thanks to c++11. Actually, there are two simpler ways - shared_ptr and unique_ptr.

shared_ptr

shared_ptr is a template which implements a class which behaves like a reference counted pointer. However, unlike a pointer, you don't have to worry about copying or destroying it; this is all handled under the hood. Each time you copy or assign a shared_ptr, it increments a reference count. Each time a shared_ptr goes out of scope, it decrements it's reference count. Lets look at some pseudo code to give you an idea:

```
template <classname T>
class SharedPtr {
    T* ptr;
     int* cnt;
    SharedPtr(T* i_ptr) : ptr(i_ptr) {
        cnt = new int(1);
     }
    SharedPtr(const SharedPtr& p) :
    ptr(p.ptr) cnt(p.cnt)
    {
        (*cnt)++;
    }
    SharedPtr* operator=(const SharedPtr& p) {
        if (this == &p)
            return *this;

        ptr = p.ptr;
        cnt = p.cnt;
```

```
        *cnt++;
        return *this;
    }
    ~SharedPtr() {
        (*cnt)--;
        if (*cnt  <= 0) {
            delete ptr;
            ptr = nullptr;
            *cnt = 0;
        }
    }
}
```