RAII

RAII stands for Resource Acquisition is Initialization. The idea is that you eliminate "naked new operations" and instead handle all new'ing in constructors and all deletion in destructors, keeping memory management the domain of well defined abstractions. The STL embodies this example (as does the last implementation of Person), and we have already come in contact with a class which embodies this idea - std::vector.

We are going to go over the Standard Template Library in some detail in later lessons, but, for now, lets look a bit deeper at the vector. First, we need to get a question out of the way: why do we need std::vector? Why don't we have an array as part of c++? Well the answer is that we *do* have array in C++; however, array size must be fixed at *compile time* according to the C++ Standard. Here is what native array usage looks like in C++:

```
#include <iostream>

using namespace std;

int main() {
    int nums[10];
    for(int i=0; i< 10; i++) {
        nums[i] = i+1;
    }

    for(auto i:nums) {
        cout << i << endl;
    }
    return 0;
}
```

This type of array in C++ is stored in a contiguous block of memory on the stack which is sized to hold the declared number of types. If you declare an array of 10 ints, and an int is 4 bytes then the compiler will set asside exactly 40 bytes for the array. No more, no less; it cannot grow or shrink after the compilation. Native arrays in C++ are far more restrictive than in Python, whose array is effectively like the std::vector; it resides on the heap and may be any size you like. It will grow periodically in response to append calls (again like vector).

C++ authors have addressed this deficiency by providing an abstraction which behaves like a dynamic array in the STL. And it is simple to use. The STL authors have embraced RAII and completely abstracted memory management for you. You don't have to worry about vector having to allocate additional memory, copy existing memory into the new location, de-allocation, etc. All you have to do is call `push_back` to add an element to the end of the array.

```
#include <vector>
```

```
#include <iostream>
//
uisng namespace std;
//
int main() {
    vector<int> nums;

    for(int i=0; i<10;i++) {
        nums.push_back(i+1);
    }

    for (auto i : nums ) {
        cout << i << endl;
    }
    return 0;
}
```

When approaching problems in C++ which require you to dynamically manage memory, you should consider the same approach.

C++11 Memory Management

The good news is, C++11 has added a couple of pointer wrappers which handle alot of this for you out of the gate. These two wrappers are: shared_ptr and unique_ptr.

shared_ptr

shared_ptr is a template 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.

Since we have not talked about templates yet, I will give you a quick rundown. A template is a mechanism provided by C++ to allow you to define type variables. Let's say you have a function which has a calling parameter and you want to implement that function for a variety of types. Instead of having to define the function for each type, you can use a template to declare a *type variable*, and then refer to the type variable instead of each concrete type. You have already used templates before actually. Recall that when we employed *vector*, we had to enclose the type of the vector in carots (<>). That is how you use a template. Now we are going to use one. Don't worry if you don't quite get this; we will spend a whole session on templates later...

Getting back to shared_ptr, lets look at an approximate implementation to give you an idea of what its about:

```
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;

        // decrement the current count
        (*cnt)--;
        // redirect shared_ptr to the new one
        ptr = p.ptr;
        cnt = p.cnt;
        *cnt++;
        return *this;
    }
    ~SharedPtr() {
        (*cnt)--;
        if (*cnt  <= 0) {
            delete ptr;
            ptr = nullptr;
            delete cnt;
            cnt = nullptr;
        }
    }
}
```

The cool thing about using shared_ptr in place of a raw pointer within a class is that you no longer have to implement a copy constructor, assignment operator, or destructor, as the memory management is handled by the shared_ptr class. Lets see how this works with Person.

Here is the .h file, in which we have also provided an implementation. A lot simpler than main.:

```
#include <memory>
class Person{
```

```
    std::shared_ptr<std::string> firstname;
    std::shared_ptr<std::string> lastname;
 public:
    Person(const std::string& fn, const std::string& ln) :
    firstname(std::make_shared<std::string>(fn)),
    lastname(std::make_shared<std::string>(ln))
    {};

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

};
```

This is great. But there is a catch. As we hand around the shared_ptr, we end up with multiple references to the same data. To illustrate this, lets go back and set our `firstname` and `lastname` to public as well. Simply move the `public:` specificer to the top, making everything public in Person. Now let's use it.

```
#include "PersonSharedPtr.hpp"
#include <iostream>


int main () {
    Person person("Frank", "Ford");
    Person person2 = person.
    person.firstname = "Fred";
    person.greet();
    person2.greet();
    return 0;
}
```

unique_ptr