Advanced C++ Programming: CS179

Lecture 12-1: Smart Pointers

Recap

- A program that wants to do more than one thing "at a time" uses threads
  - In quotes because nothing is literally simultaneous
- A thread is a side program that can do something that takes time without making the main program wait for it
- Detach, yield, and join control how they are made and used
- A "race condition" is when two threads want to use the same data
  - x is 7. One thread wants to read and another wants to change it to 9. What does the first see?

Memory Leaks

```
// Allocate a rock on heap
Rock *X = new Rock;

// Pointer moves to new allocation location, old rock is lost
X = new Rock;

// Only frees the second one
delete X;
```

- The "heap" is what we call memory that is not part of our exe that we ask the OS for dynamically
- You can have multiple pointers point at a single object
- If you have zero pointers pointing at something, you can never find it again
  - That's a memory leak
- If you leak all of your computer's memory, the OS will shut you down

Scope

```
if( true )
{
        Rock X;// Made here
}// Destroyed at the closing
bracket

X.height = 0;// Out of scope

Rock *Y = new Rock;// made
delete Y; // destroyed
```

- "A variable only exists inside its closest set of curly brackets"
- With an object, when it goes out of scope it is destroyed
- With a pointer, the variable is just the pointer, not the heap memory for the object
- Can we use Scope to help with memory leaks?

Scope Destructors

```
template< typename T>
class SPointer{
        T* data;
public:
        Helper(T * X)
                data = X;
        ~Helper()
                delete data;
};
Dog *A = new Dog;
SPointer B(A);
```

- This first idea is a primitive form of smart pointers and solves leaks
  - New C++ versions have keywords for this
- A Decorator is something that adds functionality to its inner class
- The Decorator is a class too, so it has its own scope, constructor, and destructor rules
- So this makes a dog as normal, then the decorator adds a custom destructor.
- When the SPointer goes out of scope, the dog is deleted and you can't forget

With Operators

```
template<typename T>
class SPointer {
        T* data;
public:
        SPointer(T*);
        ~SPointer();
        T& operator *()
                return *data;
        T* operator ->()
                return data;
};
SPointer A(new Dog);
(*A).Bark(); A->Bark();
```

- In the last slide, we made the dog and the SPointer separately because we wanted to still use the "A" pointer later
- With operator overloading of the SPointer itself, we can make it less fragile
  - Fragile = Code that only works if you use it the way the programmer intended
  - Helper = still calling it an SPointer so I can save "Smart Pointer" for the modern version
- Only one variable now - the SPointer
  - Treat it as if it were a Dog pointer

auto_ptr

```
class auto_ptr {
T*data;
auto_ptr& operator =
        (auto_ptr &other)
{
       data = other.data;
       other.data = nullptr;
       return *this;
}
};// Only one auto ptr can be in charge of any one object
```

- The first real attempt from C++ to make this official was auto_ptr
  - auto_ptr has since been deprecated for the more powerful keywords below
- SPointer above has a flaw in that more than one of them could be assigned to the same memory
  - They can't both delete the same thing. Crash.
- Auto_ptr adds an operator =, so that only one auto_ptr can point at one object
  - Act of setting me does a clear on you

unique_ptr

```
auto_ptr<Dog> A(new Dog);
auto_ptr<Dog> B;
B = A;
// Looks like they are the same now, but A has become blank and given up control.  I hate operator overloads, reason 435.

unique_ptr<Dog> C(new Dog);
unique_ptr<Dog> D;
```

- The modern version of the deprecated auto_ptr is 99% the same
- Only change is that it doesn't allow the = operator, and replaces it with the C++11 "move" command
  - This is really a bug fix. auto_ptr couldn't handle arrays, and they couldn't be used in STL containers. (auto_ptr overloaded = in a way that no longer meant =.)
    - They don't end up the same. It's a transfer
- All the concepts are the same though
- We'll talk more about "move" in general next week in the C++ version lecture

shared_ptr

```
shared_ptr<Dog>
        A(new Dog);
shared_ptr<Dog> B;
B = A;// Two now pointing at the dog

A = nullptr;// Dog still here since B is holding on to him

B = nullptr; // Now the dog is deleted
```

- A "Reference count" is a number that tracks how many different entities are using something
    - Customers in a store are a refcount. Add one when they arrive, subtract one when they leave. When it hits 0, you can close the store
- If five shared_ptrs are pointing at the same object, that object will get deleted as soon as the fifth lets go
    - This is exactly like Java, except faster
    - A unique_ptr is a shared_ptr of max 1

weak_ptr

```
shared_ptr<Dog>
        A(new Dog);
shared_ptr<Dog>B;
B = A; // 2 refcount

weak_ptr<Dog>C;
C = A; // still 2

A = nullptr; B = nullptr;
C is now null
```

- shared_ptr is useful when a bunch of variables want to own the same object
  - Five shared_ptrs to a network connection, let it go when all 5 finish
- Sometimes you want a looser connection. Aggregation vs Association
  - An object displaying the state of the connection. You don't want it to keep the connection alive, but it does want to know if it is closed
- A weak_ptr doesn't count towards the recount of the target, but it is set to null if that target hits refcount 0
  - Passive watcher instead of owner

Converting weak to shared

```
shared_ptr<Dog>
        A( new Dog );
weak_ptr<Dog> B;
B = A;// still refcount 1

shared_ptr<Dog> C;
C = B.lock();// now it is 2
```

- You might have a weak pointer to something and then want to make it shared
- The "lock" method promotes a weak to shared
  - There isn't one for unique. Using = with unique transfers ownership, not adds ownership

The "make_*" Functions

```
unique_ptr<Dog>
    A(new Dog);

unique_ptr<Dog> B;
B = make_unique
    (new Dog);
```

- All of these examples have used constructors to make the smart pointers
- This looks like a code style issue, because sites seem to use one or the other consistently

Documentation

- Since we are outside the book for the rest of the semester, here are some reading links for more information:
  - https://docs.microsoft.com/en-us/cpp/cpp/smart-pointers-modern-cpp?view=vs-2019
  - https://www.geeksforgeeks.org/smart-pointers-cpp/
  - http://www.cplusplus.com/reference/memory/unique_ptr/
  - https://www.nextptr.com/tutorial/ta1358374985/shared_ptr-basics-and-internals-with-examples