C++: Memory Problems

or

When Good Memory Goes Bad

Announcement

- RIT Career Fair
  - Thursday, Oct 3rd 1pm – 7pm
  - Friday, Oct 4th 9am – 5pm (interviews)
- Clark Gym
- www.rit.edu/co-op/careers

Announcement

- Date for Exam 1 has been changed!
  - New date: Monday Oct 7th

Project

- Questions?
- Everyone have a partner?
- Please e-mail me with the name of your partner and I will assign you a group account.
- **Design diagrams due Oct 1st !!!!**

Plan for today

- Memory Woes
  - Memory Leaks
  - Pointer Ownership
  - Dangling Reference
  - Overwriting Arrays
- "It's a pointer problem"

Memory Leak

- A bug in a program that prevents it from freeing up memory that it no longer needs.
- As a result, the program grabs more and more memory until it finally crashes because there is no more memory left.
- In short:
  - Allocating without cleaning up.

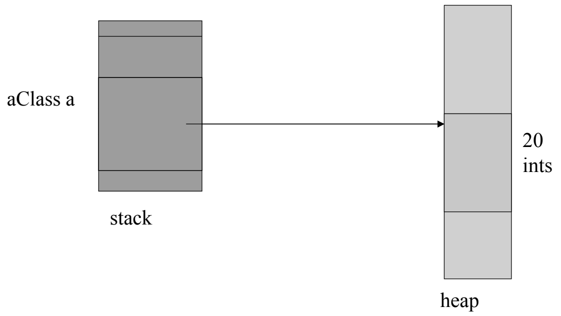
Memory Leak

```
class Foo
{
private:
    int *array_member;
    int asize;
    ...
public:
    Foo (int size);
    ~Foo ();
}
```

Memory Leak

```
Foo::Foo (int size) :
 asize (size), array_member (new int[size])
{
  for (int i=0; i<size; i++)
     array_member[i] = 0;
}

void f ()
{
  // local aClass object
  aClass a (20);
  ...
}
```

Memory Leak



Pointer Ownership

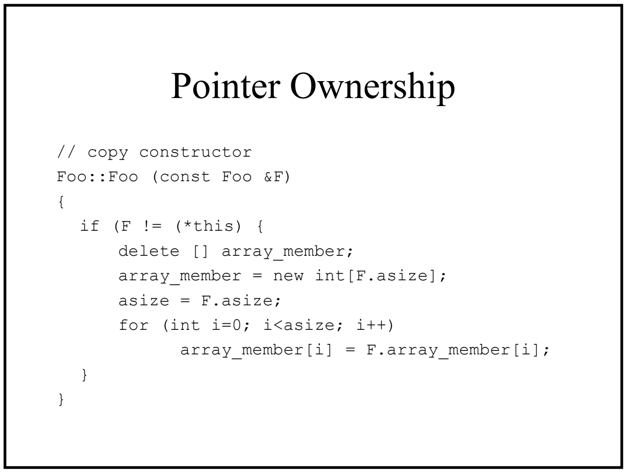
- Everything that is a pointer should be owned
  - Responsible for cleanup when finished
  - Should be known to programmer
  - Should be by design during implementation.
  - Owner and only owner should perform a delete.

Pointer Ownership

- Class members
  - If you allocate it during construction, you should deallocate during destruction.
  - Should also deallocate during
    - Copy construction
    - `operator=`

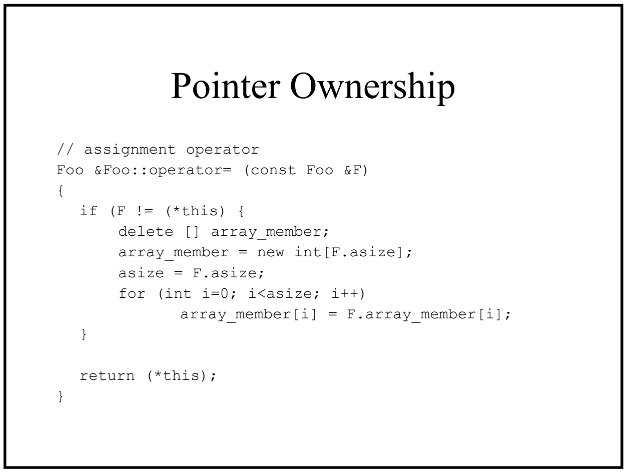
Pointer Ownership

```
// constructor
Foo::Foo (int size) :
 asize (size), array_member (new int[size])
{
  for (int i=0; i<size; i++) array_member[i] = 0;
}

// destructor
Foo:~Foo ()
{
  delete [] array_member;
}
```



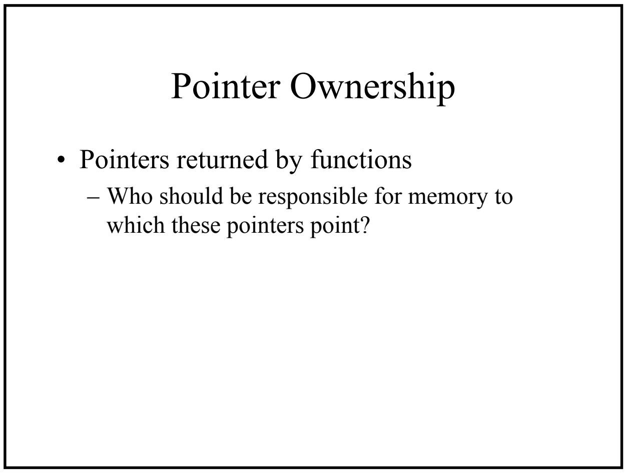
Pointer Ownership

```
// copy constructor
Foo::Foo (const Foo &F)
{
  if (F != (*this) {
      delete [] array_member;
      array_member = new int[F.asize];
      asize = F.asize;
      for (int i=0; i<asize; i++)
              array_member[i] = F.array_member[i];
  }
}
```



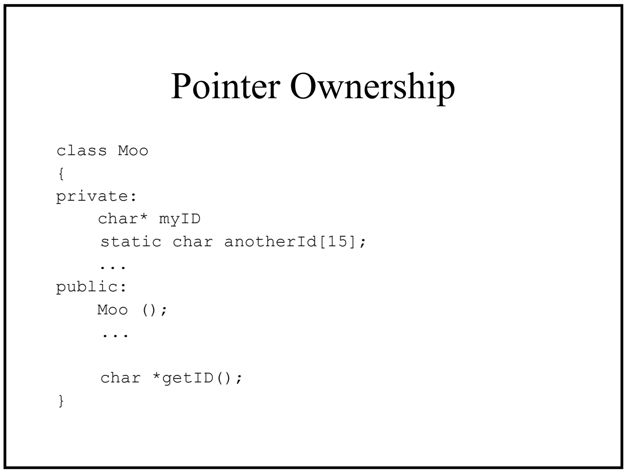
Pointer Ownership

```
// assignment operator
Foo &Foo::operator= (const Foo &F)
{
  if (F != (*this) {
      delete [] array_member;
      array_member = new int[F.asize];
      asize = F.asize;
      for (int i=0; i<asize; i++)
              array_member[i] = F.array_member[i];
  }

  return (*this);
}
```



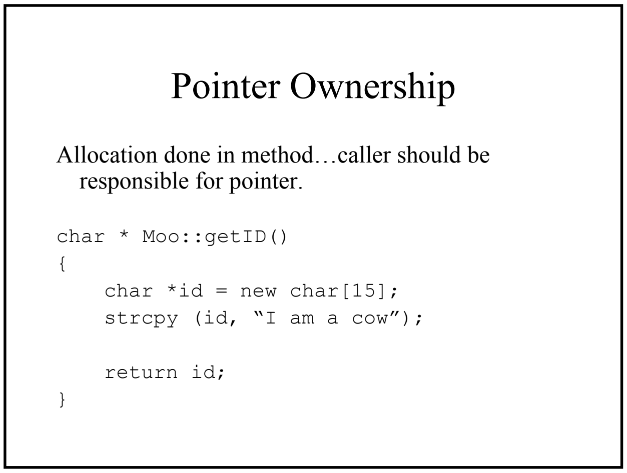
Pointer Ownership

- Pointers returned by functions
  - Who should be responsible for memory to which these pointers point?



Pointer Ownership

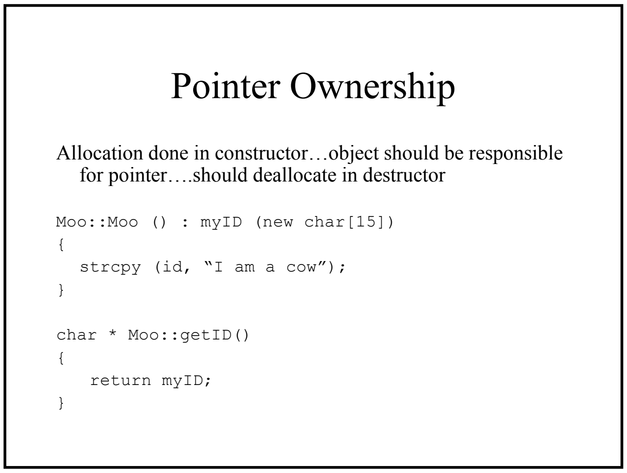
```
class Moo
{
private:
     char* myID
     static char anotherId[15];
     ...
public:
     Moo ();
     ...

     char *getID();
}
```



Pointer Ownership

Allocation done in method…caller should be responsible for pointer.

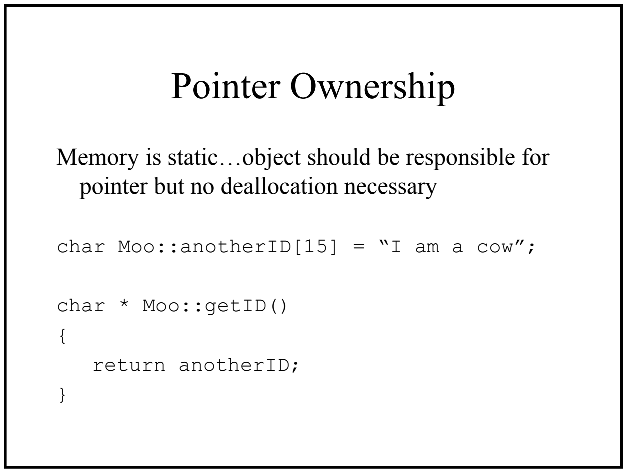
```
char * Moo::getID()
{
    char *id = new char[15];
    strcpy (id, “I am a cow”);

    return id;
}
```



Pointer Ownership

Allocation done in constructor…object should be responsible for pointer….should deallocate in destructor

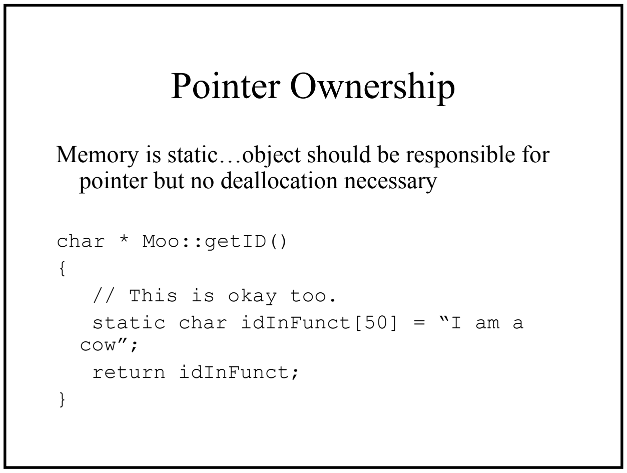
```
Moo::Moo () : myID (new char[15])
{
  strcpy (id, “I am a cow”);
}

char * Moo::getID()
{
   return myID;
}
```



Pointer Ownership

Memory is static…object should be responsible for pointer but no deallocation necessary

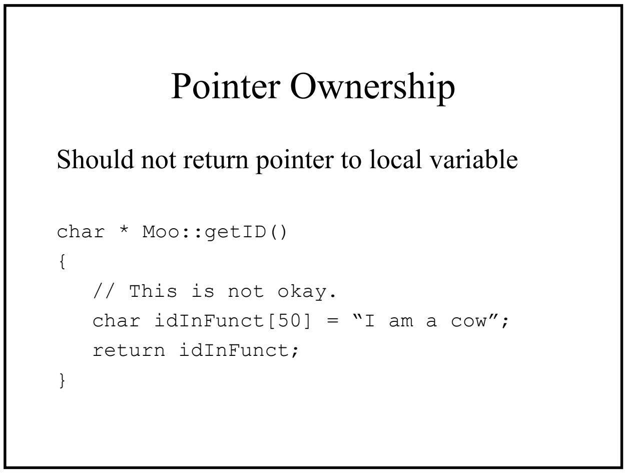
```
char Moo::anotherID[15] = “I am a cow”;

char * Moo::getID()
{
   return anotherID;
}
```



Pointer Ownership

Memory is static…object should be responsible for pointer but no deallocation necessary

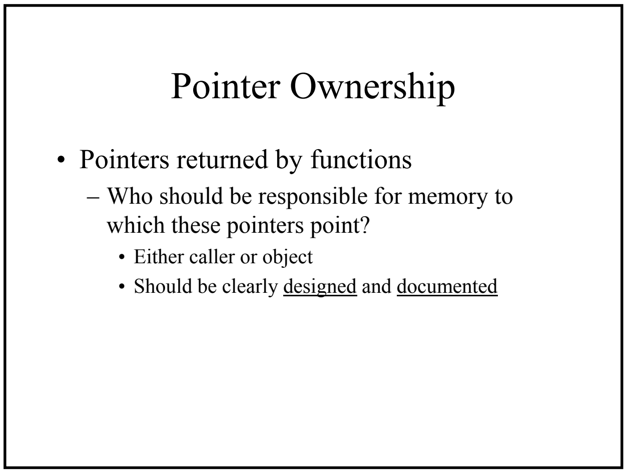
```
char * Moo::getID()
{
   // This is okay too.
   static char idInFunct[50] = “I am a
  cow”;
   return idInFunct;
}
```



Pointer Ownership

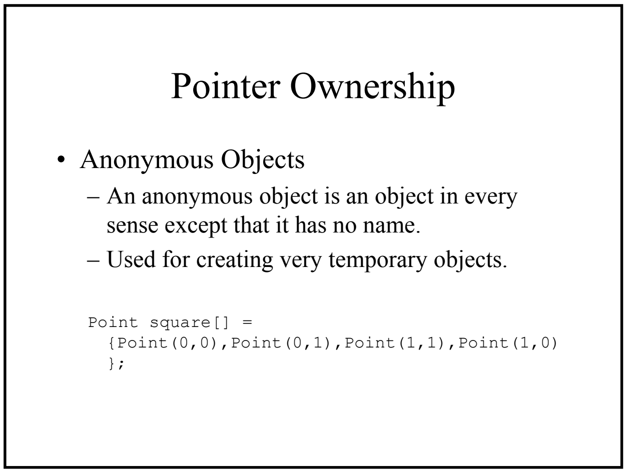
Should not return pointer to local variable

```
char * Moo::getID()
{
   // This is not okay.
   char idInFunct[50] = “I am a cow”;
   return idInFunct;
}
```



Pointer Ownership

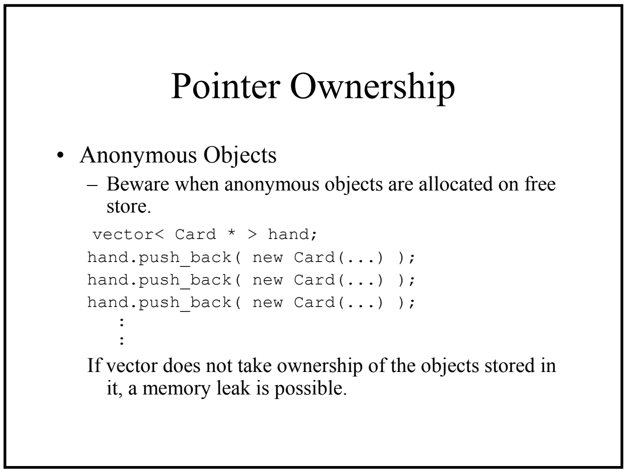
- Pointers returned by functions
  - Who should be responsible for memory to which these pointers point?
    - Either caller or object
    - Should be clearly designed and documented



Pointer Ownership

- Anonymous Objects
  - An anonymous object is an object in every sense except that it has no name.
  - Used for creating very temporary objects.

```
Point square[] =
  {Point(0,0),Point(0,1),Point(1,1),Point(1,0)
  };
```



Pointer Ownership

- Anonymous Objects
  - Beware when anonymous objects are allocated on free store.

```
 vector< Card * > hand;
hand.push_back( new Card(...) );
hand.push_back( new Card(...) );
hand.push_back( new Card(...) );
     :
     :
```

If vector does not take ownership of the objects stored in it, a memory leak is possible.

Memory Leak / Pointer Ownership

- Questions?

Dangling Pointers

- Pointer is pointing to something that it shouldn't be.
- Can happen if:
  - If the scope of a pointer extends beyond that of the object being pointed to
    - i.e Returning a pointer to a local variable.
  - If a dynamically allocated memory cell is freed explicitly and then the pointer pointing to such a space is used in subsequent code.

Dangling Pointers

```
p1 = new Foo;
  :
delete p1;
  :
p2 = new Bar; // What if same memory is
              //given to p2?
  :
int i = p1->data;  // i contains garbage
p1->op(...); // p2's object mysteriously
  changes!
```

Dangling Pointers

- Ways to prevent dangling pointers
  - Do not return pointers to local variables.
  - After calling `delete` on a pointer, immediately set it to NULL.

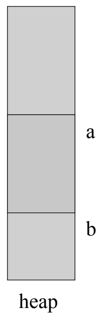
```
p1 = new Foo;
delete p1;
p1 = 0;
p2 = new Bar;
p1->op(...); // core dump!
```

Overwriting Arrays

- Recall that C++ has no array bounds checking.
  - Writing past bounds of array will trash unsuspecting memory.

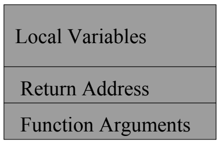
Overwriting Arrays

```
void foo()
{
   int *a = new int[20];
   aClass *b = new aClass();
   ...
   a[20] = 23; // b
  mysteriously
               // changes

}
```



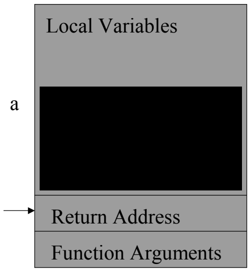
Overwriting Arrays

- This can be quite dangerous for local arrays:



Overwriting Arrays

```
void foo()
{
   int a[20];
   a[20] = 23;


}
```



Overwriting Arrays

```
void foo()
{
   char a[20];
   strcpy (a, “This string is too long”);
}
```

Dangling Pointers / Overwriting Arrays

- Questions?

Getting around these problems

- The smart pointer
  - Prevents memory leaks and dangling pointers
  - Wrapper class that owns a pointer to an object
    - Object keeps a reference count of variables accessing it
    - When the reference count reaches 0, the object is deleted by the smart pointer.
    - After deleting object, pointer value set to 0.

Getting around these problems

- The smart pointer

- But more on this tomorrow…

- Questions?