Memory Management Techniques

How can we ensure we clean up after ourselves?
You Have Two Choices

1. Dynamically allocated objects are “owned” by some other object who is responsible for deleting them.

2. Dynamically allocated objects are responsible for deleting themselves when nobody is using them anymore.
How have we been doing this?
Composition Makes Things Easy

Document “owns” Actors
When Document is destroyed, it destroys any actors.

```cpp
CDocument::~CDocument()
{
    for(list<CActor *>::iterator a=mActors.begin(); a!=mActors.end(); a++)
    {
        delete *a;
    }
    mActors.clear();
}
```
Can we make some of this automatic?

**Smart Pointers** – A class that contains a pointer that can be used as if it were a pointer, but automatically destroys the object when it is destroyed.
std::auto_ptr<>

```cpp
#include <memory>

int main(int argc, _TCHAR* argv[]) {
    auto_ptr<int> a(new int);
    *a = 22;
    cout << *a << endl;

    auto_ptr<int> b;

    b = auto_ptr<int>(new int);
    *b = 7;

    cout << *b << endl;
    return 0;
}
```

auto_ptr<T> works mostly like T*, except it *automatically* deletes what is points to when it is destroyed.
Another example

**without auto_ptr**

```cpp
void CClass::QueryAndUse()
{
    CDBQuery *query(new CDBQuery);
    if(!query->DoQuery())
    {
        delete query;
        return;
    }
    query->Process();
    delete query;
}
```

**with auto_ptr**

```cpp
void CClass::QueryAndUse()
{
    auto_ptr<CDBQuery> query(new CDBQuery);
    if(!query->DoQuery())
    {
        return;
    }
    query->Process();
}
```

*Note: No *
auto_ptr and classes

class CA
{
    public:
        CA() : mB0(new CB) {mB1 = auto_ptr<CB>(new CB);}
        void SetB2(CB *b) {mB2 = auto_ptr<CB>(b);}

    private:
        auto_ptr<CB> mB0;
        auto_ptr<CB> mB1;
        auto_ptr<CB> mB2;
};

When CA is destroyed, the objects pointed to by mB0, mB1, and mB2 will automatically be destroyed.
auto_ptr<> semantics

Can construct with a pointer, but no assign a pointer.

```cpp
auto_ptr<int> b;
b = auto_ptr<int>(new int);
*b = 7;
```

```cpp
auto_ptr<int> c;
c = b;
```

What is pointed to by b is now pointed to be c. The pointer passed to c and b now points to nothing (null).

Only one auto_ptr object can own the pointer
auto_ptr<> warnings

Cannot be used a template classes where the class may make copies of the pointer. In other words: don’t use in STL containers.

```
vector<auto_ptr<CDrawable>> mDrawablesInOrder;
```

auto_ptr semantics can be confusing, so many programmers avoid it.
Boost libraries

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared_ptr&lt;T&gt;</td>
<td>pointer to T&quot; using a reference count to determine when the object is no longer needed. shared_ptr is the generic, most versatile smart pointer offered by boost.</td>
</tr>
<tr>
<td>scoped_ptr&lt;T&gt;</td>
<td>a pointer automatically deleted when it goes out of scope. No assignment possible, but no performance penalties compared to &quot;raw&quot; pointers</td>
</tr>
<tr>
<td>intrusive_ptr&lt;T&gt;</td>
<td>another reference counting pointer. It provides better performance than shared_ptr, but requires the type T to provide its own reference counting mechanism.</td>
</tr>
<tr>
<td>weak_ptr&lt;T&gt;</td>
<td>a weak pointer, working in conjunction with shared_ptr to avoid circular references</td>
</tr>
<tr>
<td>shared_array&lt;T&gt;</td>
<td>like shared_ptr, but access syntax is for an Array of T</td>
</tr>
<tr>
<td>scoped_array&lt;T&gt;</td>
<td>like scoped_ptr, but access syntax is for an Array of T</td>
</tr>
</tbody>
</table>

Boost libraries tend to be the “future” of C++ libraries
Things are not always so simple, though

A directed graph consists of nodes and edges between the nodes. In this case the edges have weights.
Ideas on the Memory Management?

Suppose the graph is deleted. How do we delete all of the nodes and edges?
One Approach

DirectedGraph simply owns all of the nodes and edges

A common approach when you have a collection of things that are all related. How would you implement? Do you need the back links?
What about deleting nodes?

What should happen when I delete a node?
What about deleting nodes?

Can I just delete all from-edges then and all to-edges in a loop?

What if a node has an edge to itself?
Another Case

Remember the Observer Pattern?
Subjects and Observers

When should we delete the subject?

```
Subject
AttachObserver(in observer : Observer)
DetachObserver(in observer : Observer)
UpdateObservers(in ignore : Observer = NULL)
```
Can we delete when there are no more observers?

Is an Observer the only thing that may have a pointer to our subject?

What was the approach used in Project 2?
A Common Problem

What if I use the same image in more than one place?
One Approach

Since an image can be shared, we’ll put it in its own object. Then picture can have an association with it.

But, can we delete Image when we delete Picture?
One Approach

We can’t delete Image when we delete Picture because it may be used by another Picture

But, it has to get deleted eventually!
Reference Counters

When we link to an object, we call `IncRef()`. When we unlink, we call `DecRef()`. When the number of references is zero, nothing is pointing at `Image`, so it can delete itself.
Example Code

class CImage
{
    public:
        CImage() {mRefs = 0;}
    virtual ~CImage() {};

    void IncRef() {mRefs++;}
    void DecRef() {mRefs--; if(mRefs == 0) delete this;}

    wxImage *GetImage() {return &mImage;}

    private:
        wxImage mImage;
        int mRefs;
};
void CPicture::SetImage(CImage *image)
{
    // Indicate we will point to this image
    image->IncRef();

    // If we are pointing at something else, release it
    if(mlImage != NULL)
    {
        mlImage->DecRef();
        mlImage = NULL;
    }

    mlImage = image;
}

CSE 335 Dr. Charles B. Owen
Object-Oriented Programming
Base classes?

class CImage
{
public:
    CImage() {mRefs = 0;}
    virtual ~CImage() {}

    void IncRef() {mRefs++;}
    void DecRef() {mRefs--; if(mRefs == 0) delete this;}

    wxImage *GetImage() {return &mImage;}

private:
    wxImage mImage;
    int mRefs;
};
If you use this much...

class CRefCount
{
    public:
    virtual ~CRefCount();

    void IncRef() {mRefs++;}
    void DecRef() {mRefs--; if(mRefs == 0) delete this;}

    protected:
    CRefCount() {mRefs = 0;}
};

A reference counter abstract base class. Just derive something from this that needs a reference counter.
Caveats

When using reference counters, all instances **must** be dynamically allocated.

*Don’t do this:*

```cpp
CImage image;
picture1->SetImage(&image);
picture2->SetImage(&image);
```

*Instead:*

```cpp
CImage *image = new CImage();
picture1->SetImage(image);
picture2->SetImage(image);
```
Reference Cycles

Reference counters won’t help if you have circular references. If A points and B and B points to A, both will always have a reference count of 1.
Reference Counters are very popular

In wxWidgets if you do this:

```cpp
wxImage a(L"images/whatever.png")
wxImage b = a;  // This does not actually copy the image
```

Microsoft uses reference counters for COM (Component Object Model) objects. This is how DirectX, Direct3D, and many other MS API’s work.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QueryInterface</td>
<td>Retrieves pointers to the supported interfaces on an object.</td>
</tr>
<tr>
<td>AddRef</td>
<td>Increments the reference count for an interface on an object.</td>
</tr>
<tr>
<td>Release</td>
<td>Decrements the reference count for an interface on an object.</td>
</tr>
</tbody>
</table>

Python uses reference counters for objects.

```python
void shutDown( void )
{
    if( g_pd3dDevice != NULL )
        g_pd3dDevice->Release();
    if( g_pD3D != NULL )
        g_pD3D->Release();
}
```
Biggest Problem with Reference Counters

Remembering to IncRef()/DecRef() all of the time!

Can we make this automatic?
More Smart Pointers

A pointer inside an object. The object allows us to be “smart” about how we use the pointer.

When we set the pointer, automatically IncRef().
When we clear the pointer, automatically DecRec().
Smart Pointer Example

template <class T> class CPtr
{
public:
    CPtr() {mPtr = NULL;}
    CPtr(T *pPtr) {mPtr = pPtr; if(mPtr) mPtr->IncRef();}
    CPtr(const CPtr &pPtr) {mPtr=pPtr.mPtr; if(mPtr) mPtr->IncRef();}
    ~CPtr() {Clear();}

    void Clear() {if(mPtr) {mPtr->DecRef(); mPtr = NULL;}}
    T *operator=(T *t) {if (t) t->IncRef(); Clear(); mPtr = t; return mPtr;}
    T *operator=(CPtr &t)
        {if (t.mPtr) t.mPtr->IncRef(); Clear(); mPtr = t.mPtr; return mPtr;}
    operator T *() const {return mPtr;}
    T *operator->() const {return mPtr;}

private:
    T *mPtr;
};

CPtr<CImage> img = new CImage();
picture1->SetImage(img);
picture2->SetImage(img);
Other Smart Pointers

CComPtr<> : Microsoft smart pointer for COM
wxObject is effectively a smart pointer
CIteratorPtr is a smart pointer that automatically deletes an iterator
Boost library smart pointers