Memory Management (II)

Object Counting

- "Object counting" is one technique for avoiding memory leaks
- When the program starts, initialize the object count to zero
- Every time an object is constructed, increment the object count
- Every time an object is destructed, decrement the object count
- Just before the program terminates, verify that the object count is zero

Object Counting

```
int objectCount = 0;

class A {
public:
   A() { ++objectCount; … }
   ~A() { --objectCount; … }
   …
};

class B {
public:
   B() { ++objectCount; … }
   ~B() { --objectCount; … }
   …
};

void main() {
   …
   cout << "Object Count: " << objectCount << endl;
}
```

Object Counting

- Adding code to manage the object count to every class is tedious
- It is convenient to put this code in a base class from which other classes may inherit this functionality

```
class ObjectCount {
private:
   static int creations;
   static int deletions;
public:
   ObjectCount() { ++creations; }
   ~ObjectCount() { ++deletions; }
   static int GetCreations() { return creations; }
   static int GetDeletions() { return deletions; }
   static int GetObjectCount() { return creations-deletions; }
   …
};
```

Object Counting

```
#include "ObjectCount.h"

class A : public ObjectCount {
public:
   A() { … }
   ~A() { … }
   …
};

class B : public ObjectCount {
public:
   B() { … }
   ~B() { … }
   …
};

void main() {
   …
   cout << "Object Count: " <<
           ObjectCount::GetObjectCount() <<
           endl;
}
```

Object Counting

- If the object count isn't zero at the end of the program, how do we fix it?
- To figure out where the leak is, we need to know what kinds of objects aren't being freed
- The ObjectCount class tells us that there's a memory leak, but it doesn't help us figure out which objects are being leaked
  - ObjectCount only keeps a single global counter
- If there are dozens of classes in the program, how can we determine the types of objects that are being leaked?

Object Counting

- In addition to the global counter, we can also keep track of object counts on a per-class basis
- If the global count indicates that there is a memory leak, we can then query each class individually for its object count
- This tells us what kinds of objects are being leaked, and gives us some clues about where the problem might be
- The ObjectCountBase and ObjectCount classes from the CS240 Utilities provide global and per-class object counts

Object Counting Utilities

```
#include "ObjectCount.h"

class A : public ObjectCount<A> {
Public:
   A() { … }
   ~A() { … }
   …
};

class B : public ObjectCount<B> {
Public:
   B() { … }
   ~B() { … }
   …
};

void main() {
   …
   if (ObjectCountBase::GetGlobalObjectCount() != 0) {
      cout << "A: " << ObjectCount<A>::GetClassObjectCount() << endl;
      cout << "B: " << ObjectCount<B>::GetClassObjectCount() << endl;
   }
}
```

Resource Management

- Memory isn't the only kind of resource that must be carefully managed
- Other kinds of resources that can be allocated and freed include:
  - Files
  - Network connections
  - GUI resources - windows, widgets, fonts, cursors, etc.
  - Database connections
- These resources are allocated and freed using OS system calls
- Any of them can be leaked if they aren't properly freed

Error Conditions & Resource Leaks

- Resource leaks are especially likely when errors occur
- Your code should ensure that dynamically-allocated resources are ALWAYS freed, not just when everything goes well

```
char * buffer = new char[data_size];
ifstream file("somefile");
if (!file) {
   cout << "Could not open file" << endl;
   return;
}
// read data into buffer
// process the data
delete [] buffer;
```

Error Conditions & Resource Leaks

- Does this code have a potential resource leak?

```
char * buffer = new char[data_size];
DoSomething(buffer);
delete [] buffer;
```

- Yes! If DoSomething throws an exception, buffer is never deleted
- How do we solve this type of problem in Java?

```
FileReader file;
try {
   file = new FileReader("somefile");
   DoSomething(file);
}
finally {
   file.close();
}
```

Error Conditions & Resource Leaks

- C++ doesn't have "finally", so how do we solve this type of problem in C++?
- Destructors
- Whenever you dynamically allocate a resource, wrap it in an object whose destructor frees the resource
- Destructors are always called when an object goes out of scope, even when a function "returns" or an exception is thrown

```
class CharArrayDeallocator {
private:
   char * array;
public:
   CharArrayDeallocator(char * a) { array = a; }
   ~CharArrayDeallocator() { delete [] array; }
};
```

Error Conditions & Resource Leaks

```
char * buffer = new char[data_size];
CharArrayDeallocator cad(buffer);
ifstream file("somefile");
if (!file) {
   cout << "Could not open file" << endl;
   return;
}
// read data into buffer
// process data
//delete [] buffer;
```

```
char * buffer = new char[data_size];
CharArrayDeallocator cad(buffer);
DoSomething(buffer);
//delete [] buffer;
```

Error Conditions & Resource Leaks

- This style of programming prevents resource leaks, but it's a little awkward
- The next step is to add methods to the wrapper class so that all access to the resource is performed through the object itself
- Example: ifstream

```
int CountWords(const string & fileName) {
   int count = 0;
   string word;
   ifstream file(fileName);
   while (true) {
      file >> word;
      if (file) {
         ++count;
      }
      else {
         return count;
      }
   }
}
```

File is automatically opened by the ifstream constructor

File is automatically closed by the ifstream destructor (You don't have to call file.close, but you can if you want to)

Smart Pointers

- A common example of wrapping dynamically-allocated resources in objects is "smart pointers"
- Smart pointers are like regular pointers, except they automatically delete the referenced object when they go out of scope

```
void DoStuff() {
   Widget * w = new Widget();
   w->DoSomething();
   w->DoSomethingElse();
   cout << *w << endl;
}
```

Memory leak! We never deleted w, and our only pointer to it has been lost

Smart Pointers

- C++ provides a smart pointer class named auto_ptr that helps us avoid this common programming error
- #include <memory>
- Use auto_ptr<Widget> instead of Widget *

```
#include <memory>
using namespace std;

void DoStuff() {
   auto_ptr<Widget> w = new Widget();
   w->DoSomething();
   w->DoSomethingElse();
   cout << *w << endl;
}
```

No memory leak. The smart pointer automatically deletes the object when it goes out of scope

Smart Pointers

- Notice that we are able to use the -> and * operators on our smart pointer, just like with regular pointers

```
void DoStuff() {
   auto_ptr<Widget> w = new Widget();
   w->DoSomething();
   w->DoSomethingElse();
   cout << *w << endl;
}
```

- Why does this work?
- The auto_ptr class overloads the -> and * operators

Smart Pointers

- auto_ptr also has a copy constructor and operator =

```
void DoDifferentStuff() {
   auto_ptr<Widget> w = new Widget();
   auto_ptr<Widget> x = w;
   auto_ptr<Widget> y;
   y = x;
   …
}
```

- Why does this code work? Doesn't it try to delete the same object three times?
- No. The auto_ptr copy constructor and operator = transfer ownership of the object from one auto_ptr to another so that only one of them will delete it (w and x are null by the time their destructors are called)

Reference Counting Utilities

- auto_ptr is great, but it's only useful when there's just one reference to an object
- With reference counted objects, there can be many references to an object
- We want to delete a reference counted object only when the last reference has gone away
- The CS240 Utilities provide a smart pointer class that works with reference counted objects

Reference Counting Utilities

- To make a reference counted class, subclass the Referencable base class
- Referencable stores a reference count and provides AddRef and ReleaseRef methods for managing the reference count

```
#include "Referencable.h"

class Widget : public Referencable { … };

void DoStuff() {
   Widget * w = new Widget();
   w->AddRef();
   w->DoSomething();
   DoSomethingElse(w);
   if (w->ReleaseRef() == 0) {
      delete w;
   }
}
```

Manually managing reference counts is extremely error-prone

Reference Counting Utilities

- Rather than managing reference counts manually, use the Reference smart pointer class

```
#include "Referencable.h"
#include "Reference.h"

class Widget : public Referencable { … };

void DoStuff() {
   Reference<Widget> w = new Widget();
   w->DoSomething();
   DoSomethingElse(w);
}
```

The Reference constructor automtically calls AddRef

The Reference destructor automtically calls ReleaseRef and deletes the object if the count becomes zero