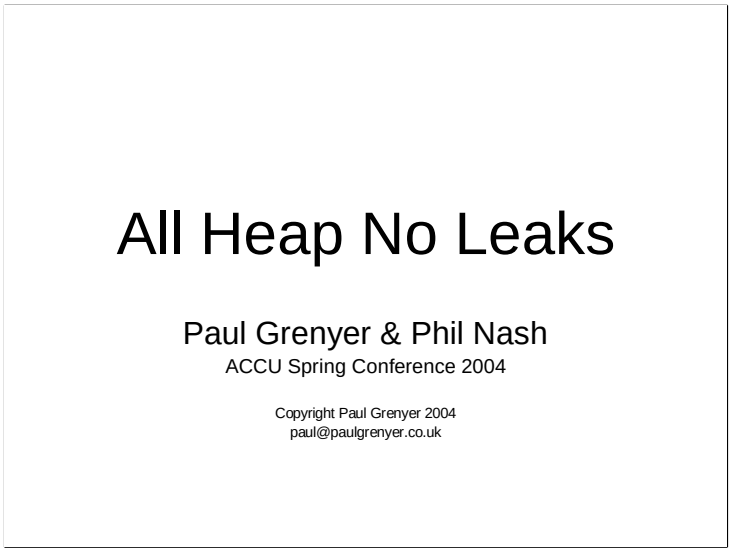
Slide 1



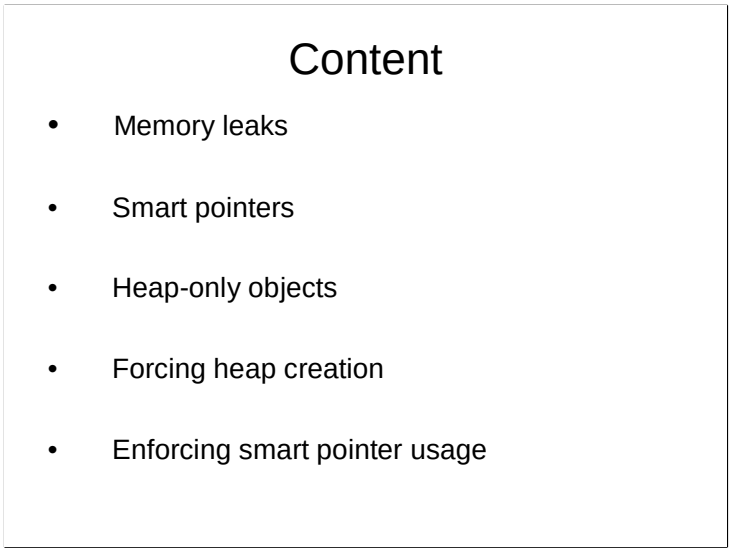
- Me & Phil Nash
- Problems caused by new and delete in C++ for novice and expert developers
- New without corresponding delete equals a memory leak
- C#, Python and Java have a garbage collector

The use of new and delete in C++ can cause problems for both novice and expert developers alike. Using new without a corresponding delete results in a memory leak. Some languages such as C#, Java and Python provide managed objects that can be created on the managed heap leaving deletion to the garbage collector.

- Smart pointers in C++ allow similar behaviour with deterministic destructor
- User must know to use smart pointer, many don’t
- C++ objects only on heap and managed by smart pointers

Smart pointers allow similar behaviour to be achieved in C++, with the added bonus that they provide deterministic destruction of the object to which they point. However, the user still has to know that they should be using smart pointers and many don't. In this talk I am going to look at a way of writing C++ objects that can only be created on the heap, and *must* be managed by (memory management) smart pointers.

Slide 2



- Memory leaks
- Smart pointers
- Heap-only objects
- Forcing heap creation
- Enforcing smart pointer usage

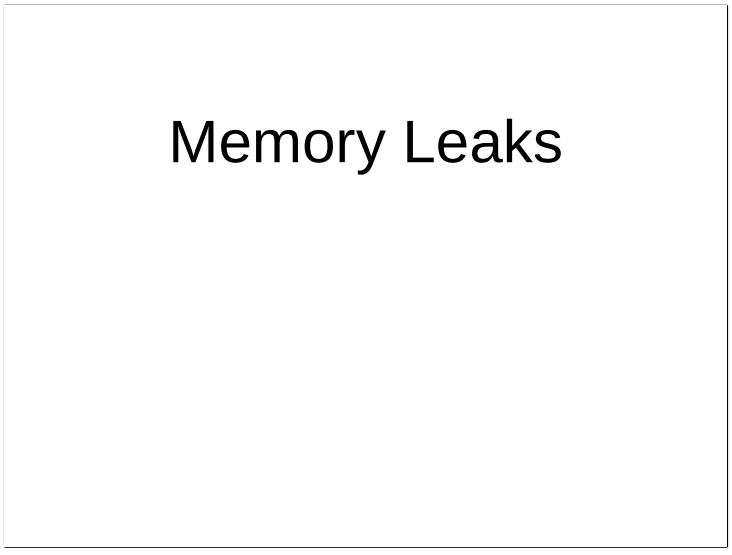
I am going to start by looking at how memory leaks occur.

Then I'll introduce smart pointers and look at the implementation of a *very* simple smart pointer, similar to boost's scoped pointer.

Following on from that I'll look, very briefly, at what sort of objects should be restricted to heap-only creation and then how to enforce heap only creation.

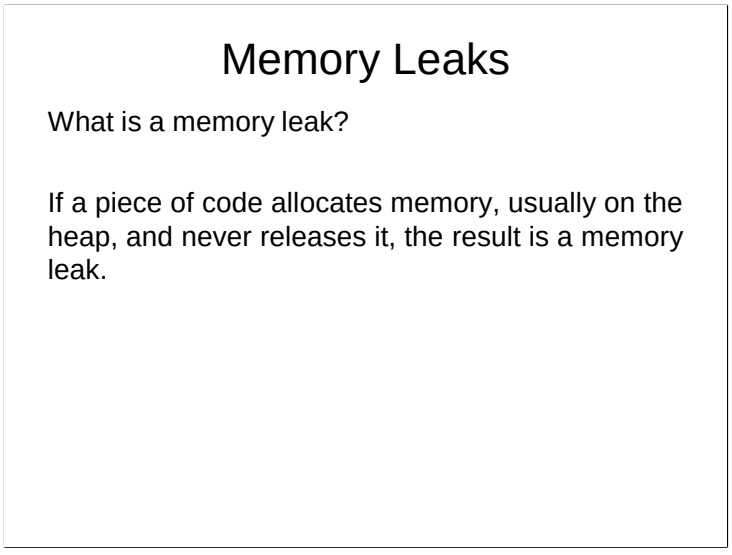
Finally, I'll examine how the use of a smart pointers can be enforced when using a heap-only object.

Slide 3



Let’s start by looking at memory leaks.

Slide 4



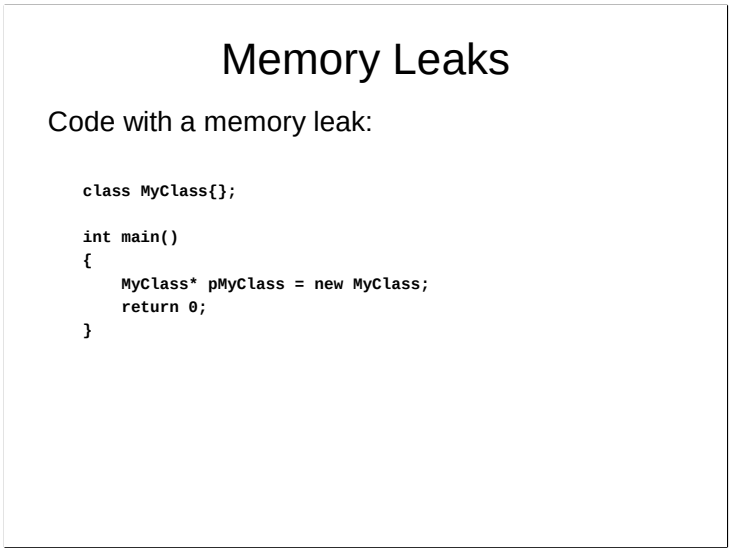
- New without a corresponding delete.
- Stack and Heap (sutter)

As already stated, the most common cause of a memory leak is a call to new without a corresponding call to delete, but what is actually happening?

Stack — The stack stores automatic variables. Objects are constructed immediately at the point of definition and destroyed immediately at the end of the same scope.

Heap — The heap is a dynamic memory allocated using new and freed using delete.

Slide 5



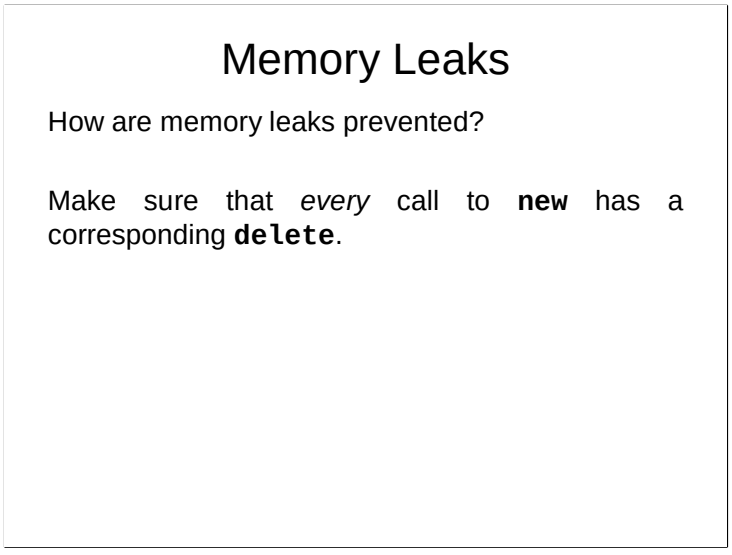
- `MyClass` created on the heap, referenced by a pointer
- Pointer is on the stack
- Pointer is destroyed at the end of `main()` scope
- Memory allocated on the heap is lost

The above example shows an instance of the `MyClass` object being created on the heap, using the new key word. A pointer is used to reference the object.

The pointer is created on the stack. When the pointer goes out of scope at the end of `main()`, it is destroyed, but the `MyClass` object it references and the memory in which it was created remains on the heap, until the process ends, where it may or may not be cleaned up by the operating system.

The memory allocated to hold the `MyClass` instance is effectively lost by the program. This is referred to as a memory leak.

Slide 6

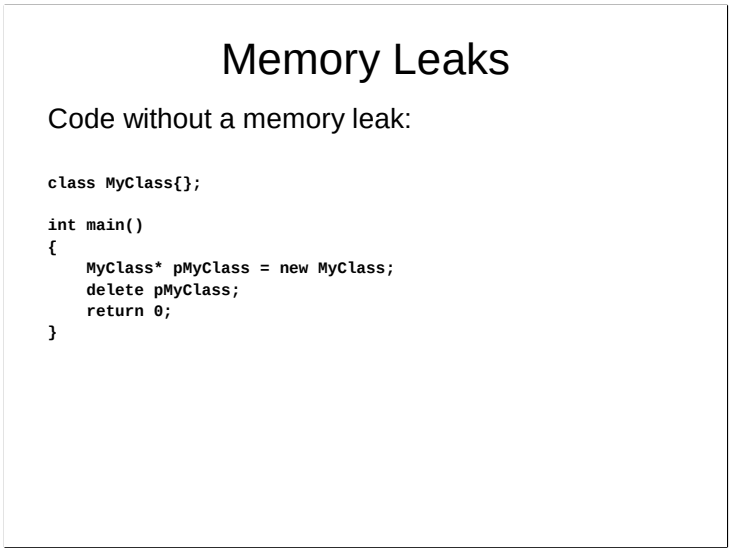


- Keyword `new` used to allocate memory on the heap
- Keyword `delete` used to free memory allocated on the heap
- Therefore, make sure every call to `new` has a corresponding `delete`

The key word new is used to create objects on the heap and the delete key word is used to destroy them and free the memory from the heap.

Therefore *most* memory leaks can be avoided by making sure that every call to new has a corresponding call to delete.

Slide 7



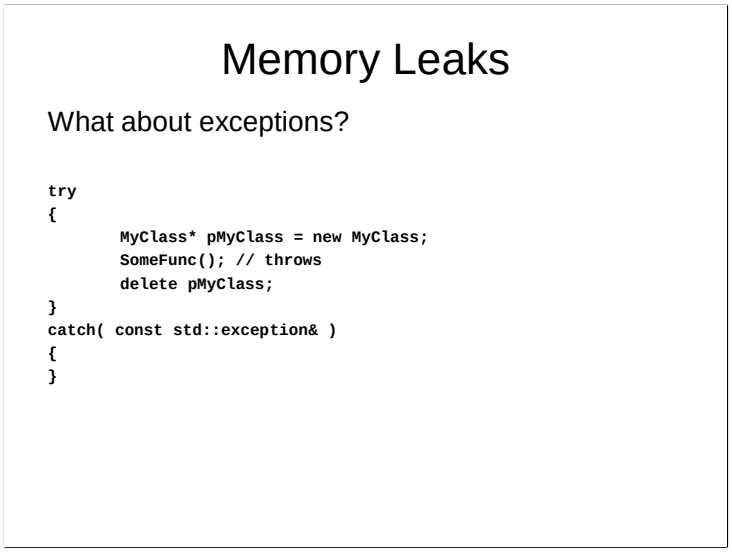
- `MyClass` object created on the heap, referenced with a pointer.
- `MyClass` object freed using `delete`, before pointer goes out of scope
- No memory leak!

The code, again, shows an instance of the `MyClass` object being created on the heap, using the new key word and it is still referenced by a pointer that is created on the stack.

However, there is now a call to delete following the call to new. This means that the `MyClass` object is now destroyed and the memory it was created in released before the pointer goes out of scope at the end of main.

No more memory leak!

Slide 8



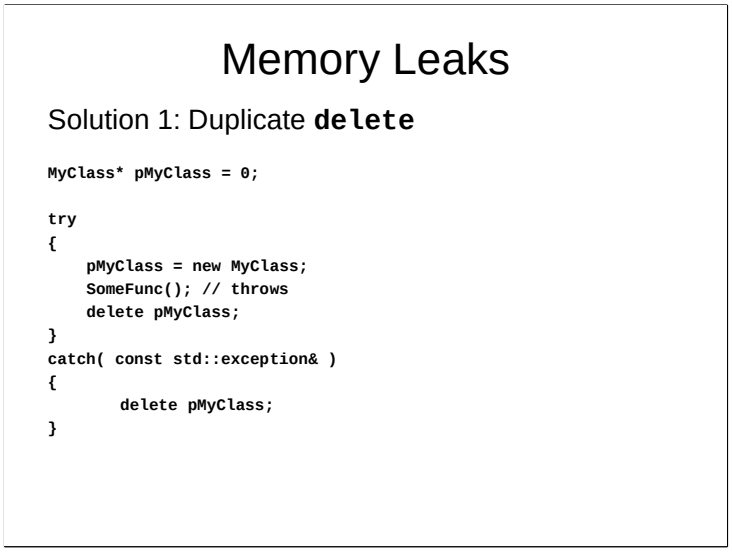
- Modern C++ is full of exceptions
- Example shows new with corresponding `delete`
- Exception causes memory leak

What about exceptions? Modern C++ is full of exceptions and they can indirectly lead to memory leaks.

The code shows a `MyClass` object being created on heap by a call to new and there is a corresponding delete, so there shouldn't be a memory leak, right? Wrong!

After the call to new there is a call to the `SomeFunc` function. Somewhere in the `SomeFunc` function an exception is thrown. At that point the stack starts to unwind. Therefore the pointer referencing the `MyClass` object goes out of scope and the call to delete is never reached. As before, this causes a memory leak.

Slide 9



- One solution, move pointer declaration outside of `try/catch` and duplicate `delete`
- Call delete in both `try` and `catch` block
- Unnecessary code duplication
- Relies on catch, catching the exception thrown by `SomeFunc()`.

One solution is to move the declaration of the `MyClass` pointer out of the scope of the try block and call delete in the catch block as well as in the try block.

If the exception thrown by `SomeFunc()` is of a type than can be caught by the catch block, the stack stops unwinding at the end of the try block and the code in the catch block is executed.

The pointer referencing the `MyClass` object is still in scope, as it was declared outside of the try block. Therefore the call to delete can be made in the catch block and the memory freed correctly.

This causes unnecessary code duplication and assumes that the exception thrown by `SomeFunc` can be caught by the catch block. If it can't the stack continues to unwind, the pointer goes out of scope without delete begin called and the memory is still lost.

Slide 10

Memory Leaks

Solution 2: **`delete`** after **`catch`**

```
MyClass* pMyClass = 0;

try
{
    pMyClass = new MyClass;
    SomeFunc(); // throws
}
catch( const std::exception& )
{
}

delete pMyClass;
```

- Solution 2, move the `delete` to the end of the `try/catch` block.
- Removes duplication
- Still relies on the catch block catching `SomeFunc()`'s exception
- Not an ideal solution

A slightly better solution, in that it removes the duplication, is to move the delete out of the try block to below the catch block. However, this still assumes that the catch block can catch the exception thrown by `SomeFunc()`.

Slide 11

Memory Leaks

Solution 3: Use a smart pointer.

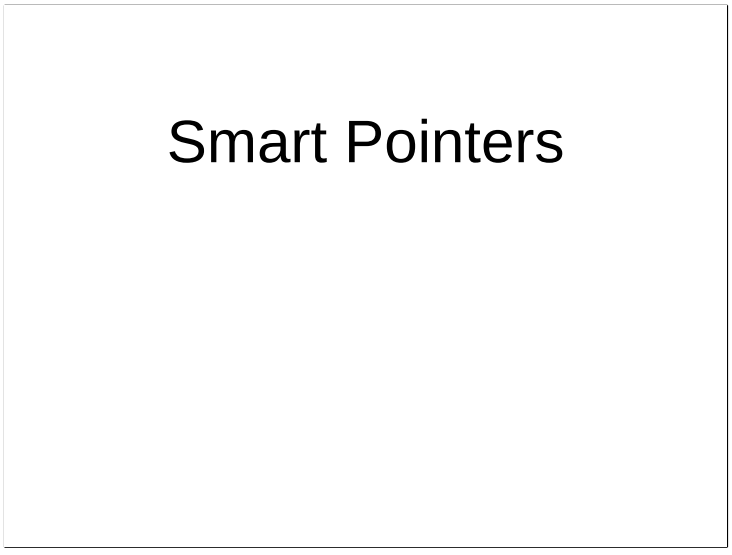
```
try
{
    SmartPtr< MyClass > pMyClass( new MyClass );
    SomeFunc(); // throws
}
catch( const std::exception& )
{
}
```

- Better if object was destroyed when pointer went out of scope?
- This is what smart pointers do

Wouldn’t it be better if the object referenced by the pointer was destroyed and the memory allocated for it was freed when the pointer went out of scope?

This is exactly what smart pointers can do.

Slide 12



So, lets’ have a closer look at smart pointers.

Slide 13

Smart Pointers

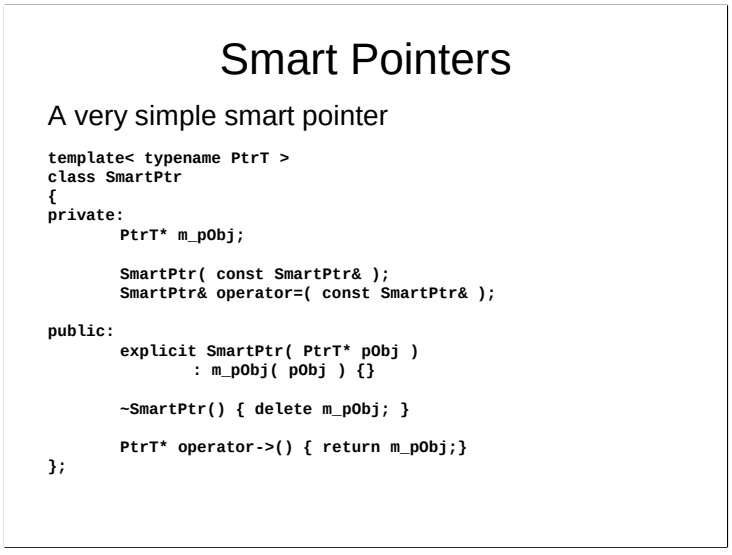
What is a smart pointer?

- A smart pointer is a class that maintains a pointer to a dynamically allocated object, and acts as if it *was* that pointer
- When a smart pointer goes out of scope, the dynamically allocated object is automatically deleted by the smart pointers destructor

A smart pointer:

- Maintains a raw pointer to a dynamically allocated object
- Acts as if it were a raw pointer
- Automatically deletes its dynamically allocated object when it goes out of scope
- Exhibits Resource Acquisition In Initialisation characteristics

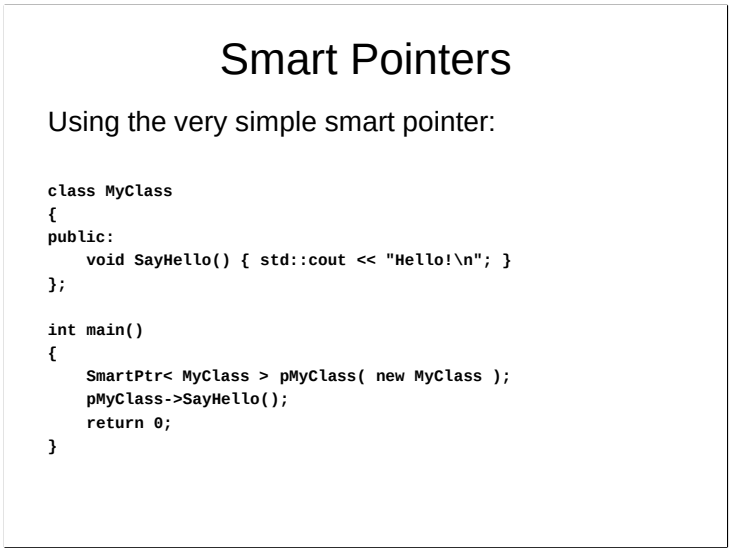
Slide 14



This is an example of a very simple smart pointer. It has been restricted to only the necessary functionality:

- It is a template so that it can be used with different types of objects
- It maintains a raw pointer to the object it is referencing
- The copy assignment constructor and copy assignment operator are private to prevent copying. Preventing copying means that it is not necessary to implement reference counting
- The object maintained by the smart pointer is passed in via the constructor and used to initialise the raw pointer data member.
- The destructor deletes the maintained object.
- An arrow operator is provided to allow the smart pointer to act as if it was a normal pointer to the maintained object.

Slide 15



- `SmartPtr` is easy to use
- A single member function has been added to `MyClass`
- The pointer returned by new is passed to `SmartPtr`
- `SayHello()` called via `SmartPtr` as if it were a raw pointer
- Call to `delete` dangerous and unnecessary

`SmartPtr`, like other smart pointers, is very easy to use as the code shows.

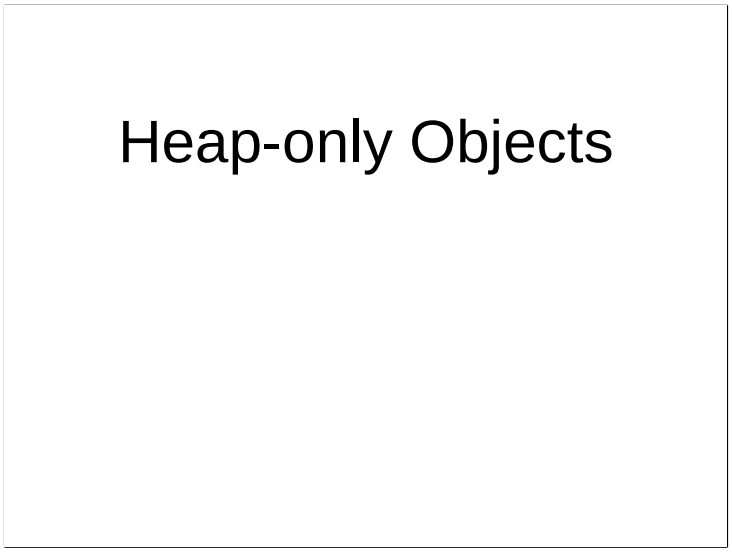
`MyClass` is a class with a single member function. The example shows `SmartPtr` being used to maintain an instance of `MyClass`:

The pointer returned by calling new to create the `MyClass` object on the heap is passed directly into the `SmartPtr` object via the constructor.

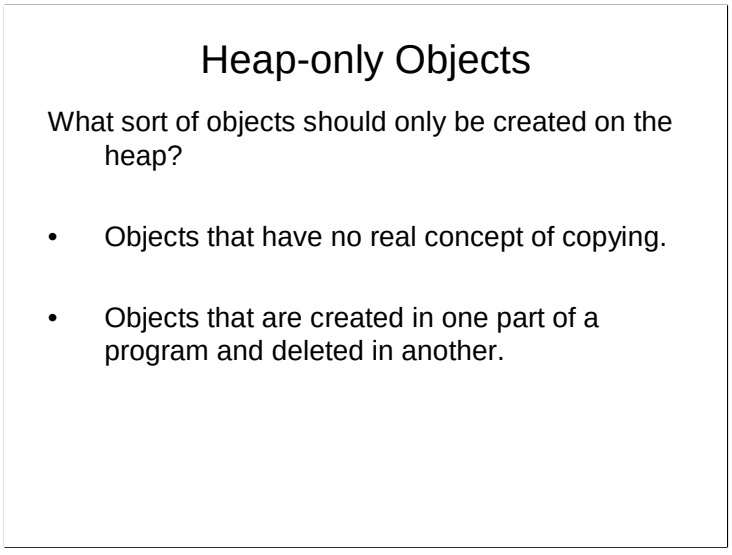
`MyClass`'s `SayHello` function is then called using the `SmartPtr` object as though it were a normal pointer.

An explicit call to delete to destroy the `MyClass` object is not required, and would in fact be dangerous, as when the `SmartPtr` object goes out of scope its destructor calls delete on the `MyClass` objects pointer, destroying it and releasing the memory.

Slide 16



Slide 17



- Resource object such as a database connection or file pointer
- No concept of copying.
- Create on heap, pass round by pointer
- Lazy evaluation
- Reference count

A resource object usually refers to a single resource such as a file pointer or a database connection.

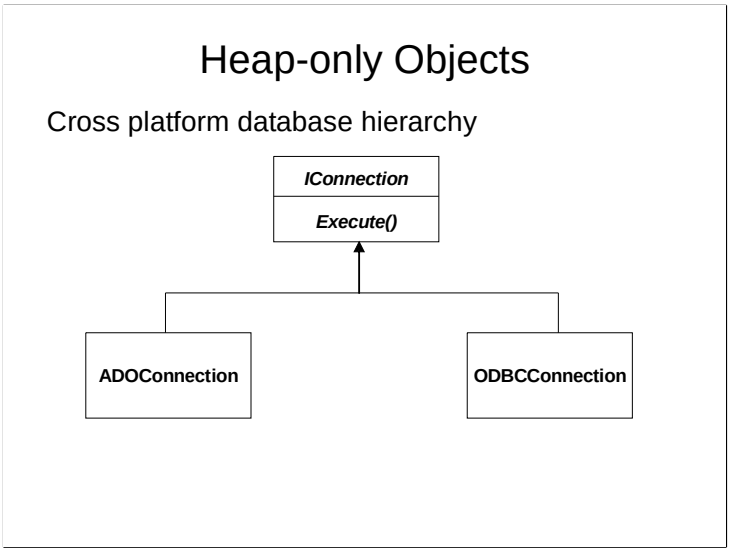
There is no real concept of copying this kind of a resource object as you would end up with two or more objects referencing the same resource. Therefore it makes sense to create a single resource object and pass references or pointers to it to those objects in the application that need to use it. This can be done more easily if the object is created on the heap.

A resource object is an ideal candidate for lazy evaluation. Opening a file or connecting to a database is an expensive operation. Therefore it should be put off until absolutely necessary and the open resource should then be maintained for as long as possible.

In an idea situation the resource object should be created, on the heap, the first time an attempt to use the resource is made. The resource object should then be passed on to other objects that need to use the resource. This can be done easily using a reference counted pointer.

If the object that originally created the resource object is destroyed, it doesn't matter as the resource object has been created on the heap and is referenced by other objects that are using the resource. As soon as the reference count goes to zero, the resource object can be destroyed.

Slide 18



- Common interface implemented by two or more objects
- Aeryn: cross-platform thin database access layer
- Use of ADO and ODBC on Windows
- Use of interface without knowing which implementation is being used
- Smart pointer of the interface type

Creating an object on the heap makes even more sense when a common interface can be implemented by two or more subclasses.

I recently developed a cross-platform thin database access layer for my C++ testing framework Aeryn.

The database access layer allows the use of both ADO and ODBC on Windows. Both ADO and ODBC use a connection string to specify the database connection and both can execute SQL statements. Therefore, both have a common interface.

The use of a common interface allows objects that use a database to be written without needed to know what type of database connection they are using.

The interface becomes the smart pointer type and maintains a heap based ADOConnection or ODBCConnection object.

Slide 19

Heap-only Objects

Cross platform database hierarchy

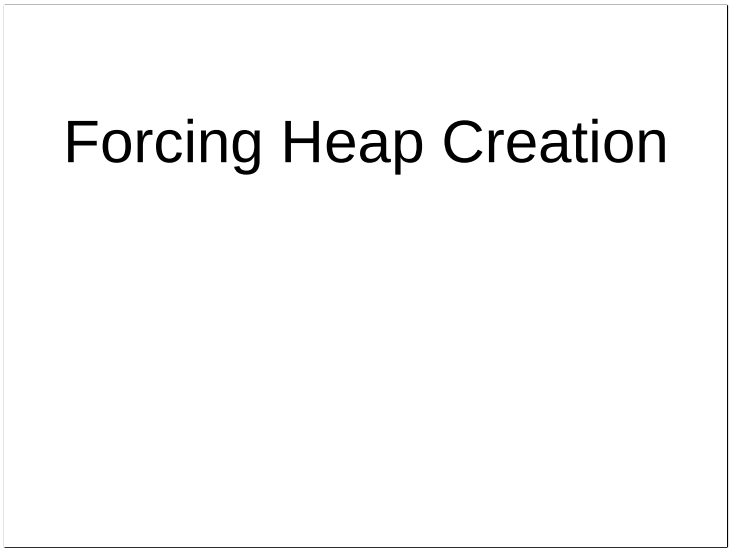
```
class IConnection
{
public:
        virtual ~IConnection() {}

        virtual void Execute( const std::string& statement ) = 0;
};


class ADOConnection : public IConnection
{
public:
        explicit ADOConnection
                ( const std::string& connectionString ) { }

        virtual void Execute
                ( const std::string& statement ) { }
};
```

- Code shows implementation of interface and concrete `ADOConnection`
- `ODBCConnection` is the same
- Connection string passed via constructor
- Execute overridden in concrete object

This code shows the declaration of the database connection interface and an empty implementation for the ADOConnection class. The ODBCConnection class definition is identical to the ADOConnection class.

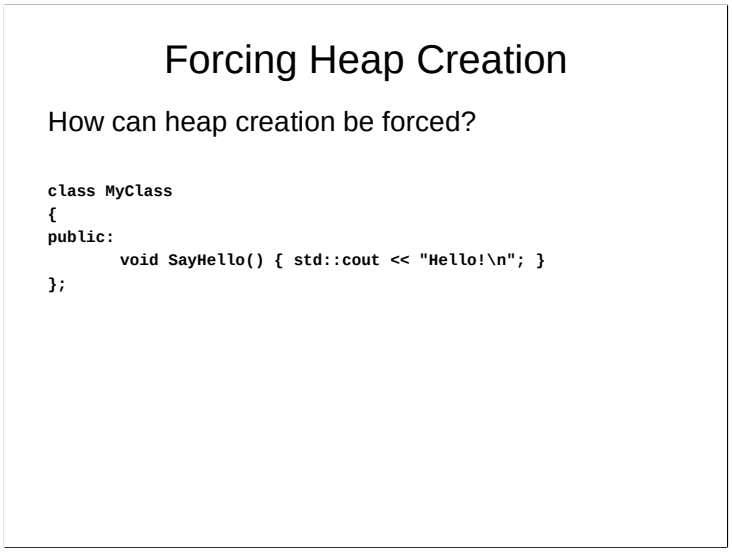
As you can see the connection string, describing the database connection is passed in through the ADOConnection constructor. The execute member function, inherited from the base class (or interface) takes a single string containing the SQL to be executed via the connection.

Slide 20



Ok, so we have just looked at the kinds of objects that should be restricted to heap creation. Let’s have a look at how heap creating can be enforced.

Slide 21



- Again, `MyClass` with single `SayHello()` function
- How can `MyClass` enforce heap creation?

The code here is the `MyClass` example from earlier, with an extra member function that simply sends “Hello!” to std out.

Let’s see how we can force this class to be created on the heap.

Slide 22

Forcing Heap Creation

Meyers: 1.Make destructor private

```
class MyClass
{
private:
        ~MyClass(){};

public:
        void SayHello() { std::cout << "Hello!\n"; }
};
```

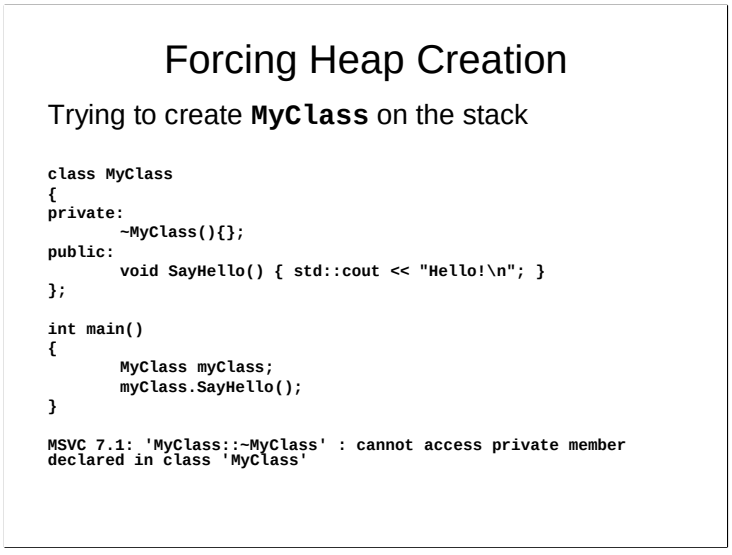
- Meyers describes a way of forcing heap-only creation in MEC++.
- Make destructor private
- Why not make constructor private?

Scott Meyers describes a way of forcing objects to be created on the heap in his book More Effective C++.

Making the destructor of a class private prevents it from begin created on the stack (or as a global or static) as it can no longer be called, except by the class itself or its friends.

The same could be achieved by making the constructor(s) private. However, this requires more maintenance as a class may have several constructors, but only one destructor.

Slide 23



- Testing Meyers with MSVC 7.1
- Cannot access destructor when creating on stack

Testing Meyers suggestion, on Microsfot Visual C++ 7.1 does indeed show that a `MyClass` object cannot be created on the stack, due to its private destructor.

Slide 24

Forcing Heap Creation

Trying to create **`MyClass`** on the heap

```
class MyClass
{
private:
        ~MyClass(){};
public:
        void SayHello() { std::cout << "Hello!\n"; }
};

int main()
{
        MyClass* pMyClass = new MyClass;
        pMyClass->SayHello();
        delete pMyClass;
}

MSVC 7.1: 'MyClass::~MyClass' : cannot access private member
declared in class 'MyClass'
```

- Can be *created* on heap with `new`
- Cannot to destroyed using `delete` as destructor is private
- Causes a memory leak!

Making the destructor private allows an object to be created on the heap, but it cannot be destroyed using delete, as delete cannot access the destructor either.

This, as discussed earlier, causes a memory leak and we don’t want that!

Slide 25

Forcing Heap Creation

Meyers: 2.Provide a disposal method:

```
class MyClass
{
private:
        ~MyClass(){};

public:
        void Destroy(){ delete this; }
        void SayHello() { std::cout << "Hello!\n"; }
};
```

- Back to Meyers…
- Add disposal method
- Disposal method calls `delete this`
- `delete this` is considered a bit risky by some people…

Ok, back to Meyers, who suggests adding a disposal method to allow a class with a private destructor to be destroyed. The disposal method can be called from outside the class as it is public and can call delete on the `MyClass` object as, as a member of the class, it has private access to the object.

However, use of `delete this` is consider by some people to be a bit risky, but it will do, for now.

Slide 26

Forcing Heap Creation

Destroying `MyCall` via a disposal method

```
class MyClass
{
private:
        ~MyClass(){};

public:
        void Destroy(){ delete this; }
        void SayHello() { std::cout << "Hello!\n"; }
};

int main()
{
        MyClass* pMyClass = new MyClass;
        pMyClass->SayHello();
        pMyClass->Destroy();
}
```

- Code shows disposal method in use
- Memory leak is prevented

The code here shows the disposal method being used to prevent a memory leak. The `MyClass` object is created on the heap, as before with by calling new. After the `SayHello()` function is called, the object is destroyed by calling the disposal method.

Slide 27

Forcing Heap Creation

What about using a smart pointer?

```
template< typename PtrT >
class SmartPtr
{
public:
        ~SmartPtr() { delete m_pObj; }
...
};

int main()
{
        SmartPtr< MyClass > pMyClass( new MyClass );
        pMyClass->SayHello();
        pMyClass->Destroy();
}

MSVC 7.1: 'MyClass::MyClass' : cannot access private member
declared in class 'MyClass'
```

- What about using a smart pointer?
- Code shows an attempt to use the `SmartPtr`
- Disposal method *can* be called explicitly, but should *not* be
- `SmartPtr`'s destructor still tries to call delete and fails
- If code worked, double delete! Bad!

The code here shows a reminder of the smart pointer shown earlier and it being used to create the new `MyClass` object.

However, although the disposal method can be called explicitly when using `SmartPtr`, the example still fails to compile as `SmartPtr` still tries to destroy the object using delete in its destructor.

Even if `SmartPtr` was able to destroy the `MyClass` object using delete, the example would try to destroy the `MyClass` object twice. This is a very bad idea!

Slide 28

Forcing Heap Creation

How can `MyClass` be used with `SmartPtr`?

- Modify `SmartPtr` to allow a function to be used instead of `delete`
- Make `MyClass`'s disposal method static so it can be called by `SmartPtr`

- Could write a smart point specifically for `MyClass`
- Modify `SmartPtr` to use a function to free objects
- Make `MyClass`' disposal method static

We are now very close to passing control of our `MyClass` object to `SmartPtr`, but we're not there yet. There are a couple more modifications we must make to both `MyClass` and `SmartPtr`.

First `SmartPtr` must be modified so that it can use `MyClass`'s disposal method, instead of just calling delete in its destructor. This could be done by writing a template specialisation of `SmartPtr` specifically for `MyClass`, but this involves extra unnecessary code, so we'll do it a slightly different way.

Although it is not obvious yet, `MyClass`'s disposal method will need to be made static so that it can be called by `SmartPtr`.

Slide 29

Forcing Heap Creation

Add a function pointer to **`SmartPtr's`** constructor

```
template< typename PtrT >
class SmartPtr
{
private:
        PtrT* m_pObj;
        void (*m_pDestroy)( PtrT* );

        SmartPtr( const SmartPtr& );
        SmartPtr& operator=( const SmartPtr& );

public:
        explicit SmartPtr( PtrT* pObj, void (*pDestroy)( PtrT* ) )
                : m_pObj( pObj ), m_pDestroy( pDestroy ) {}
        ~SmartPtr(){ m_pDestroy( m_pObj );}
        PtrT* operator->(){ return m_pObj; }
};
```

- Modify `SmartPtr` to take a second constructor param, a function pointer
- Function pointer matches signature of disposal method
- Function is called in destructor instead of `delete`
- Knock on effect is that disposal method must be static

Instead of making `SmartPtr` specific to `MyClass`, the code here shows a modification to `SmartPtr` which allows it to use any (static) function with the correct signature to destroy the object it is managing.

This is achieved by providing a function pointer member and initialising it via a constructor parameter. The function is then called in the destructor instead of delete.

This of course has the knock on effect that `MyClass`'s disposal method now has to be static.

Slide 30

Forcing Heap Creation

Adding disposal method to **`MyClass`** and testing

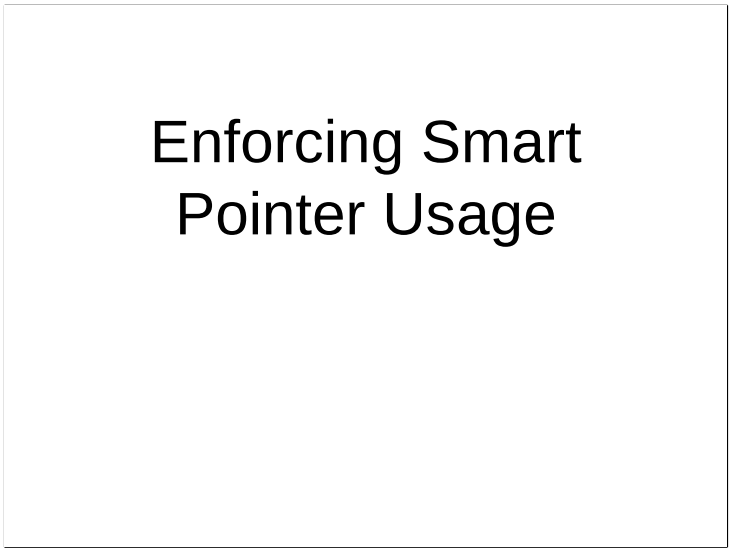
```
class MyClass
{
private:
        ~MyClass(){};

public:
        static void Destroy( MyClass* pMyClass ){delete pMyClass;}
        void SayHello() { std::cout << "Hello!\n"; }
};

int main()
{
        SmartPtr< MyClass >
                pMyClass( new MyClass, MyClass::Destroy );
        pMyClass->SayHello();
}
```

- Here `MyClass` has been modified to give it a static disposal method
- SmartPtr is not initialised with both MyClass instance and disposal method
- No more memory leak!

The code here shows a slightly modified definition of `MyClass` with a static disposal method. The disposal method is still public and can therefore still be called from outside the class and even though it is static, it still has private access to the class.

Then there is an example of how to use the disposal method with the modified `SmartPtr`. The disposal method is specified as the second `SmartPtr` constructor parameter, but there is no longer any need to call the disposal method explicitly.

Slide 31



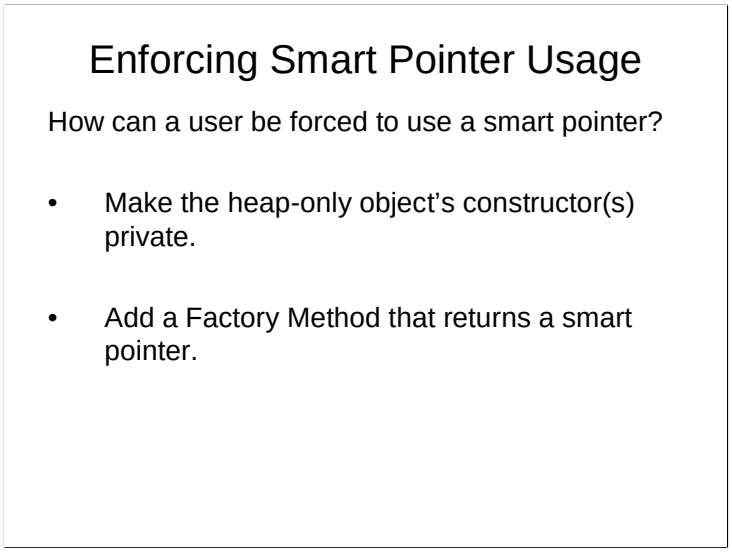
- `MyClass` can e completely managed by a smart pointer and can only be created on the heap
- It is still possible to create a `MyClass` instance without using `SmartPtr`
- Most developers don't know they should be using a smart pointer
- How can the user be forced to use a smart pointer?

So, now we have a smart pointer that can completely manage the `MyClass` object. However, it is still possible to create and correctly destroy a `MyClass` object without using `SmartPtr`.

Remember that most developers either don't know about smart pointers or don't know that they should be using them, so despite all the work that has been put into `MyClass` and `SmartPtr`, the user of `MyClass` may still not use `SmartPtr`.

Let's have a look at how we can force them to use `SmartPtr`.

Slide 32



- Compiler generated constructor allows `new` to be called
- Can be prevented by making constructor private
- A factory method must now be added

At this point `MyClass` still has the, compiler generated, default constructor that is used when a `MyClass` object is created by calling new. Calling new directly to create a `MyClass` object can be prevented by defining a default constructor and declaring it private.

However, this now prevents the object created at all, so a factory method must be added to allow an object to be created. Giving the factory method a smart pointer return type then forces the creator of the `MyClass` object to use the smart pointer.

Slide 33

Enforcing Smart Pointer Usage

Make **`MyClass`**'s constructor private

```
class MyClass
{
private:
        MyClass(){};
        ~MyClass(){};
        static void Destroy( MyClass* pMyClass )
                { delete pMyClass; }
public:
        typedef boost::shared_ptr< MyClass > MyClassPtr;

        static MyClassPtr Create()
        {
                return MyClassPtr( new MyClass, Destroy );
        }

        void SayHello(){ std::cout << "Hello!\n"; }
};
```

1. The constructor is private to force use of the factory method.
2. The destructor is private to force use of the disposal method.
3. The typedef of the smart pointer is to make use of the class easier.

(`SmartPtr` has now been abandoned. This is because it does not support any kind of copying, which is needed when using the factory method. Boost's `shared_ptr` which is now being used is reference counted and allows objects to be destroyed using a disposal method.)

4. The factory method creates an instance of the `MyClass` object and initialises the smart point with `MyClass`' disposal method and then returns a copy. This has that advantage that the user need knowing nothing about the disposal method or its usage.

Slide 34

Enforcing Smart Pointer Usage

Testing **`MyClass`**'s factory method

```
int main()
{
	MyClass::MyClassPtr pMyClass( MyClass::Create() );
	pMyClass->SayHello();
}
```

What about the Database connection?

- At last! A working solution
- The smart pointer is initialised with MyClass' factory method.
- `SayHello()` can still be called as if a raw pointer was being used
- Exception safe
- Real world example?

The example here shows how `MyClass` must now be used. As you can see `MyClass`' factory method is used to initialise a `MyClassPtr`. Explicit calling of delete or `MyClass`' disposal method are not required. There is no memory and using a `MyClass` object is now exception safe.

But how does this work in the real world, with for example, the database connection object I mentioned earlier?

Slide 35

Enforcing Smart Pointer Usage

Database connection's interface

```
class IConnection
{
public:
        static void Destroy( IConnection* pConnection )
                { delete pConnection; };

public:
        typedef boost::shared_ptr< IConnection > IConnectionPtr;

        virtual ~IConnection() {}
        virtual void Execute( const std::string& statement ) = 0;
};
```

- Base class / interface has smart pointer typedef
- Also has disposal method
- Relies on virtual destructor for proper deletion of subclass
- `Execute(…)` is still pure virtual

Let's start by looking at the Database connection hierarchy's interface. As it is the base class of all the other objects in the hierarchy the smart pointer typedef and the disposal method can be defined as part of the interface. The virtual destructor ensures that instances of the subclasses are destroyed correctly via the pointer and disposal method.

Slide 36

Enforcing Smart Pointer Usage

Concrete ADO connection

```
class ADOConnection : public IConnection
{
private:
        explicit ADOConnection
                ( const std::string& connectionString ) {}
        virtual ~ADOConnection(){};
public:
        static IConnectionPtr Create
                ( const std::string& connectionString )
        {
                return IConnectionPtr(
                        new ADOConnection( connectionString ),
                        IConnection::Destroy );
        }
        void Execute( const std::string& statement ) {}
};
```

- `ADOConnection` inherits from `IConnection` and implements `Execute(…)`
- `ADOConnection` has its own factory method
- Factory method passes `connectionString` on to constructor
- Destructor private to enforce use of disposal method
- Constructor private to enforce use of factory method

The `ADOConnection` subclass inherits from `IConnection`, implements the `Execute` member function and defines its own factory method, that creates an `ADOConnection` instance and returns `IConnectionPtr` smart pointer to it.

The constructor and destructor are private to enforce usage of the factory and disposal methods.

Slide 37

Enforcing Smart Pointer Usage

Concrete ODBC connection

```
class ODBCConnection : public IConnection
{
public:
        explicit ODBCConnection
                ( const std::string& connectionString ) {}
        virtual ~ODBCConnection(){};

        static IConnectionPtr Create
                ( const std::string& connectionString )
        {
                return IConnectionPtr(
                        new ODBCConnection( connectionString ),
                        IConnection::Destroy );       }

        void Execute( const std::string& statement ) {}
};
```

- Implementation of `ODBCConnection` is identical
- Factory method creates an `ODBCConnection` object instead

The implementation of `ODBCConnection` is the same as `ADOConnection`, except of course that that the factory method creates an `ODBCConnection` object instead..

Slide 38

Enforcing Smart Pointer Usage

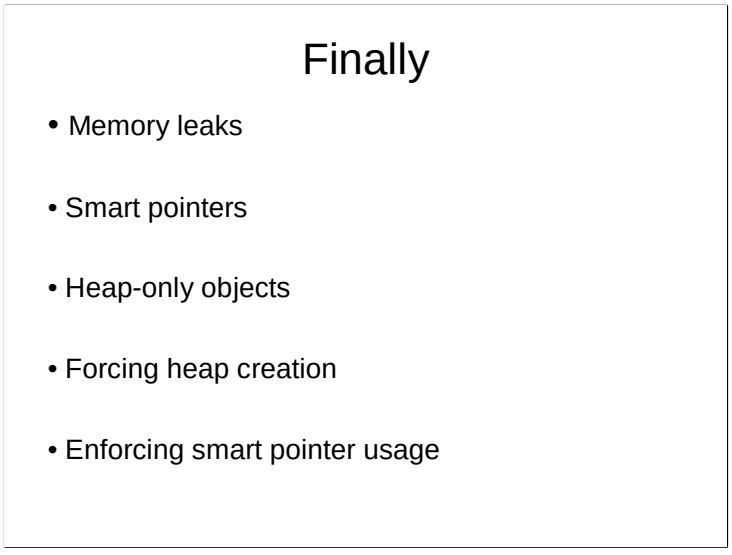
Testing ADO and ODBC connections

```
int main()
{
	typedef IConnection::IConnectionPtr IConnectionPtr;

	IConnectionPtr pAdo( ADOConnection::Create( "..." ) );
	pAdo->Execute( "..." );

	IConnectionPtr pODBC( ODBCConnection::Create( "..." ) );
	pODBC->Execute( "..." );
}
```

- Code here shows both ADO and ODBC connection objects being created and controlled via a `IConnection` smart pointer

This final piece of code shows the way in which both ADO and ODBC connection objects are created using an `IConnectionPtr` and the appropriate factory method.

Slide 39



Today we've looked at what a memory leak is and one of the common causes. Then we went on to look at what a smart pointer is and explored a very simple example. Then we had a brief look at what sort of objects should be created on the heap, followed by a way enforcing heap creation. Finally we looked at enforcing smart pointer usage and explored a very simple real world example.

Any questions?