Memory Management

Reference counting, shared pointers, buffer management, and more.

Overview

- Reference Counting
- AutoPtr, RefCountedObject and AutoReleasePool
- SharedPtr
- Buffers
- Dynamic Factory and Instantiator
- Memory Pools
- Singletons

Reference Counting

> [...] Reference counting is a technique of storing the number of references, pointers, or handles to a resource such as an object or block of memory. It is typically used as a means of deallocating objects which are no longer referenced.
>
> Wikipedia

Reference Counting (cont'd)

- Whenever a reference is destroyed or overwritten, the reference count of the object it references is decremented.
- Whenever a reference is created or copied, the reference count of the object it references is incremented.
- The initial reference count of an object is one.
- When the reference count of an object reaches zero, the object is deleted.
- In a multithreaded scenario, incrementing and decrementing/comparing reference counters must be atomic operations.

Object Ownership

- If someone takes "ownership" of an object, he is responsible for deleting the object when it's no longer needed.
- If the owner of an object fails to delete it, this will result in a memory leak.
- Others may point to the object, too, but they will never delete the object.
- Ownership is transferable. But at any given time there can only be one owner.

Reference Counting and Ownership

- A pointer taking ownership of an reference counted object does not increment its reference count.
  - This implies that there is no previous owner (in other words, the object has just been created),
  - or the previous owner gives up ownership (and thus does not decrement the object's reference count).
- Usually, the first pointer an object is assigned to after its creation takes ownership. All others do not.

The AutoPtr Class Template

- Poco::AutoPtr implements a reference counting "smart" pointer.
- Poco::AutoPtr can be instantiated with any class that supports reference counting.
- A class supporting reference counting must
  - maintain a reference count (initialized to 1 at creation)
  - implement a method void duplicate() that increments the reference count
  - implement a method void release() that decrements the reference count and, when it reaches zero, deletes the object

AutoPtr Construction and Assignment

- When constructing an AutoPtr<C> from a C*, the AutoPtr takes ownership of C (and its reference count remains unchanged).
- When assigning a C* to an AutoPtr<C>, the AutoPtr takes ownership of C (and its reference count remains unchanged).
- When constructing an AutoPtr<C> from another AutoPtr<C>, both AutoPtr's share ownership of C, and its reference count is incremented.
- When assigning an AutoPtr<C> to another AutoPtr<C>, both AutoPtr's share ownership of C, and its reference count is incremented.

```
#include "Poco/AutoPtr.h"

using Poco::AutoPtr;

class RC0
{
public:
    RC0(): _rc(1)
    {
    }

    void duplicate()
    {
        ++_rc; // Warning: not thread safe!
    }

    void release()
    {
        if (--_rc == 0) delete this; // Warning: not thread safe!
    }

private:
    int _rc;
};
```

```
int main(int argc, char** argv)
{
    RCO* pNew = new RCO;           // _rc == 1
    AutoPtr<RCO> p1(pNew);         // _rc == 1
    AutoPtr<RCO> p2(p1);           // _rc == 2
    AutoPtr<RCO> p3(pNew, true); // _rc == 3
    p2 = 0;                        // _rc == 2
    p3 = 0;                        // _rc == 1
    RCO* pRCO = p1;                // _rc == 1
    p1 = 0;                        // _rc == 0 -> deleted
    // pRCO and pNew now invalid!

    p1 = new RCO;                  // _rc == 1
    return 0;
}
                                   // _rc == 0 -> deleted
```

RefCountedObject

- Poco::RefCountedObject implements thread-safe reference counting semantics. As of 1.3.4 it uses platform-specific atomic operations, if available (Windows, Mac OS X).
- Can be used as a base class for classes implementing reference counting.
- Poco::RefCountedObject has a protected destructor and prohibits copy-construction and assignment.
- All reference counted objects should have a protected destructor, to forbid explicit use of delete.

```
#include "Poco/RefCountedObject.h"
#include "Poco/AutoPtr.h"
#include <iostream>

using Poco::RefCountedObject;
using Poco::AutoPtr;

class RCO: public RefCountedObject
{
public:
    RCO()
    {
    }

    void greet() const
    {
        std::cout << "Hello, world!" << std::endl;
    }

protected:
    ~RCO()
    {
    }
};
```

```
int main(int argc, char** argv)
{
    AutoPtr<RCO> pRCO(new RCO);

    pRCO->greet();   // AutoPtr has -> operator
    (*pRCO).greet(); // AutoPtr has * operator

    std::cout << "refcount: " << pRCO->referenceCount() << std::endl;

    RCO* p1 = pRCO; // AutoPtr supports conversion to plain pointer
    RCO* p2 = pRCO.get();

    return 0;
}
```

AutoPtr Operators and Semantics

- Poco::AutoPtr support relational operators:
  ==, !=, <, <=, >, >=
- De-referencing operators: *, ->
  will throw a NullPointerException if pointer is null
- Poco::AutoPtr supports full value semantics (default constructor, copy constructor, assignment) and can be used in collections (e.g., std::vector or std::map).
- Use AutoPtr::isNull() or AutoPtr::operator ! () to test for null

AutoPtr and Casts

- Like ordinary pointers, Poco::AutoPtr supports a cast operation.
- template <class Other>
  AutoPtr<Other> cast() const
- An AutoPtr cast is always typesafe (internally, dynamic_cast is used, so an invalid cast will result in a null pointer).
- Assignment of compatible AutoPtr's is supported through a template constructor and assignment operator.

```
#include "Poco/AutoPtr.h"
#include "Poco/RefCountedObject.h"

class A: public Poco::RefCountedObject {};
class B: public A {};
class C: public Poco::RefCountedObject {};

int main(int argc, char** argv)
{
    Poco::AutoPtr<A> pA;
    Poco::AutoPtr<B> pB(new B);

    pA = pB;                        // okay, pB is a subclass of pA
    pA = new B;

//  pB = pA;                        // will not compile
    pB = pA.cast<B>();              // okay

    Poco::AutoPtr<C> pC(new C);
    pA = pC.cast<A>();              // pA is null

    return 0;
}
```

AutoPtr Caveats and Pitfalls

- Be extremely careful when assigning an AutoPtr to a plain pointer, and then assigning the plain pointer to another AutoPtr!

- Both AutoPtr's will claim ownership of the object. This is bad!
- Explicitly tell AutoPtr that it has to share ownership of the object:
  - AutoPtr::AutoPtr(C* pObject, bool shared);
  - AutoPtr& AutoPtr::assign(C* pObject, bool shared);
  - shared must be true!

```
#include "Poco/AutoPtr.h"
#include "Poco/RefCountedObject.h"

class A: public Poco::RefCountedObject
{
};

int main(int argc, char** argv)
{
	Poco::AutoPtr<A> p1(new A);

	A* pA = p1;

//	Poco::AutoPtr<A> p2(pA);		// BAD! p2 assumes sole ownership
	Poco::AutoPtr<A> p2(pA, true);	// Okay: p2 shares ownership with p1

	Poco::AutoPtr<A> p3;
//	p3 = pA;				// BAD! p3 assumes sole ownership
	p3.assign(pA, true);			// Okay: p3 shares ownership with p1

	return 0;
}
```

AutoReleasePool

- The Poco::AutoReleasePool class template takes care of (reference counted) objects that nobody else wants.
- #include "Poco/AutoReleasePool.h"
- Poco::AutoReleasePool takes ownership of every object added to it.
- When the Poco::AutoReleasePool is destroyed (or its release() method is called), it releases the references to all objects it holds, by invoking each object's release() method.

```
#include "Poco/AutoReleasePool.h"

using Poco::AutoReleasePool;

class C
{
public:
    C()
    {
    }

    void release()
    {
        delete this;
    }
};
```

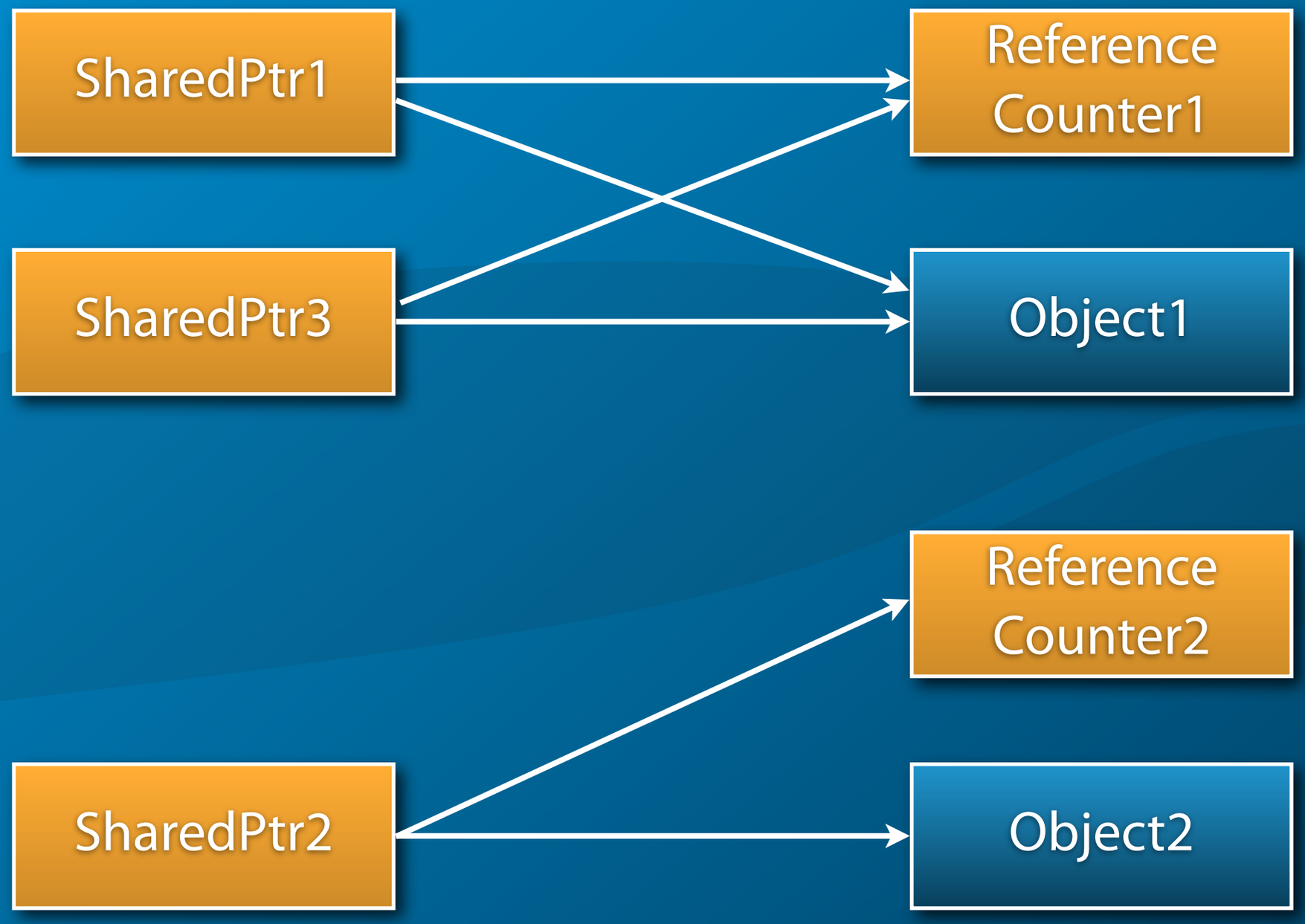
```
int main(int argc, char** argv)
{
    AutoReleasePool<C> pool;

    C* pC = new C;
    pool.add(pC);
    pC = new C;
    pool.add(pC);

    return 0;
}
// all C's deleted
```

The SharedPtr Class Template

- Poco::SharedPtr implements reference counting for classes that do not implement a reference count themselves.
- #include "Poco/SharedPtr.h"
- Poco::SharedPtr has the same features as Poco::AutoPtr (dereferencing, relational operators, etc.).
- WARNING: Assigning plain pointers to the same object to different SharedPtr's will result in multiple owners of the object causing undefined behavior (in other words, a crash).
- Once you use SharedPtr for an object, never work with plain pointers to that object again.


SharedPtr1
SharedPtr3
Reference Counter1
Object1
SharedPtr2
Reference Counter2
Object2

```cpp
#include "Poco/SharedPtr.h"
#include <string>
#include <iostream>

using Poco::SharedPtr;

int main(int argc, char** argv)
{
    std::string* pString = new std::string("hello, world!");

    Poco::SharedPtr<std::string> p1(pString);  // rc == 1
    Poco::SharedPtr<std::string> p2(p1);       // rc == 2
    p2 = 0;                                    // rc == 1
//  p2 = pString; // BAD BAD BAD: multiple owners -> multiple delete
    p2 = p1;                                   // rc == 2

    std::string::size_type len = p1->length(); // dereferencing with ->
    std::cout << *p1 << std::endl;             // dereferencing with *

    return 0;
}
// rc == 0 -> deleted
```

SharedPtr Operators and Semantics

- Poco::SharedPtr support relational operators:
  ==, !=, <, <=, >, >=
- De-referencing operators: *, ->
  will throw a NullPointerException if pointer is null
- Poco::SharedPtr supports full value semantics (default constructor, copy constructor, assignment) and can be used in collections (e.g., std::vector or std::map).
- Use SharedPtr::isNull() or SharedPtr::operator ! () to test for null

SharedPtr and Casts

- Like ordinary pointers, Poco::SharedPtr supports a cast operation.
- template <class Other>
  SharedPtr<Other> cast() const
- A SharedPtr cast is always typesafe (internally, dynamic_cast is used, so an invalid cast will result in a null pointer).
- Assignment of compatible SharedPtr's is supported through a template constructor and assignment operator.

```
#include "Poco/SharedPtr.h"

class A
{
public:
     virtual ~A()
     {
     }
};

class B: public A
{
};

class C: public A
{
};
```

```
int main(int argc, char** argv)
{
	Poco::SharedPtr<A> pA;
	Poco::SharedPtr<B> pB(new B);

	pA = pB;                          // okay, pB is a subclass of pA
	pA = new B;

//	pB = pA;                          // will not compile
	pB = pA.cast<B>();                // okay

	Poco::SharedPtr<C> pC(new C);
	pB = pC.cast<B>();                // pB is null

	return 0;
}
```

SharedPtr and Arrays

- The default implementation of SharedPtr will simply call delete pObj
- Wrong for objects created with new[]
  - need delete [] pObj;
  - create SharedPtr with a custom ReleasePolicy
    - SharedPtr<T, ReferenceCounter, ArrayReleasePolicy>

```
template <class C>
class ArrayReleasePolicy
{
public:
    static void release(C* pObj)
        /// Delete the object.
        /// Note that pObj can be 0.
    {
        delete [] pObj;
    }
};
```

```
char* pStr = new char[100];
SharedPtr<char, Poco::ReferenceCounter, ArrayReleasePolicy> p(pStr);
```

The DynamicFactory Class Template

- Poco::DynamicFactory supports object creation "by name".
- #include "Poco/DynamicFactory.h"
- All classes managed by a DynamicFactory must have a common base class. The DynamicFactory is instantiated for the base class.
- C* DynamicFactory::createInstance(const std::string& name) const;
  creates an instance of a subclass of C with the given name.
- For this to work, classes and their instantiators (factory classes) must be registered with the DynamicFactory.

The Instantiator Class Template

- An instantiator for a class C must always be a subclass of Poco::AbstractInstantiator<Base>, where Base is a base class of C.
- Poco::Instantiator is a template for a factory for a given class.
- It basically defines a method createInstance() that creates a new instance of a class, using the new operator.
- To work with class hierarchies, an Instantiator always inherits from Poco::AbstractInstantiator, which defines createInstance() for the base class.

```
#include "Poco/DynamicFactory.h"
#include "Poco/SharedPtr.h"

using Poco::DynamicFactory;
using Poco::SharedPtr;

class Base
{
};

class A: public Base
{
};

class B: public Base
{
};
```

```
int main(int argc, char** argv)
{
    DynamicFactory<Base> factory;

    factory.registerClass<A>("A"); // creates Instantiator<A, Base>
    factory.registerClass<B>("B"); // creates Instantiator<B, Base>

    SharedPtr<Base> pA = factory.createInstance("A");
    SharedPtr<Base> pB = factory.createInstance("B");

    // you can unregister classes
    factory.unregisterClass("B");

    // you can also check for the existence of a class
    bool haveA = factory.isClass("A"); // true
    bool haveB = factory.isClass("B"); // false (unregistered)
    bool haveC = factory.isClass("C"); // false (never registered)

    return 0;
}
```

DynamicFactory and Instantiators

- For the Poco::Instantiator class template to work, the class to be created must be default constructible.
- If your class is not default constructible, or if it requires special steps for construction, you must implement your own instantiator class.

```
#include "Poco/DynamicFactory.h"

using Poco::DynamicFactory;
using Poco::AbstractInstantiator;

class Base
{
};

class A: public Base
{
};

class C: public Base
{
public:
    C(int i): _i(i)
    {
    }

private:
    int _i;
};
```

```
class CInstantiator: public AbstractInstantiator<Base>
{
public:
    CInstantiator(int i): _i(i)
    {
    }

    Base* createInstance() const
    {
        return new C(_i);
    }

private:
    int _i;
};
```

```
int main(int argc, char** argv)
{
    DynamicFactory<Base> factory;

    factory.registerClass<A>("A");
    factory.registerClass("C", new CInstantiator(42));

    return 0;
}
```

Buffer Management

- When interfacing with legacy C libraries or operating system calls, one often needs to supply a buffer of a certain size.
- If the buffer is larger than a few bytes, it must be allocated on the heap.
- This requires some kind of memory management, to ensure that the buffer is deleted when it is no longer used, even in the case of an exception.
- std::auto_ptr or Poco::SharedPtr (with the default release policy) cannot be used here, because they do not work with arrays.

The Buffer Class Template

- Poco::Buffer can be used to provide a buffer (array) of fixed size that is allocated on the heap, and automatically deleted when the Buffer object gets out of scope.
- #include "Poco/Buffer.h"
- The begin() method returns a pointer to the beginning of the buffer.
- The end() method returns a pointer to the end of the buffer.
- The index operator provides access to single elements in the buffer.

```
#include <Poco/Buffer.h>
#include <string>
#include <iostream>

using Poco::Buffer;

int main(int argc, char** argv)
{
    Buffer<char> buffer(1024);

    std::cin.read(buffer.begin(), buffer.size());

    std::streamsize n = std::cin.gcount();
    std::string s(buffer.begin(), n);

    std::cout << s << std::endl;

    return 0;
}
```

Memory Pools

- Many applications allocate and release buffers of a given size very frequently.
- Allocating buffers on the heap has performance impacts, and can lead to heap fragmentation.
- Therefore it makes sense to reuse a buffer once it has been allocated.
- Poco::MemoryPool does just that.

The MemoryPool Class

- Poco::MemoryPool (#include "Poco/MemoryPool.h") maintains a collection of memory blocks of a certain size.
- A call to void* MemoryPool::get() hands out a pointer to a continous block of memory.
  - If there are no available blocks, a new block is allocated.
  - The maximum number of blocks can be limited. If no more blocks are available, a OutOfMemoryException is thrown.
- A call to void MemoryPool::release(void* ptr) releases the memory block back to the pool.

```
#include "Poco/MemoryPool.h"
#include <string>
#include <iostream>

using Poco::MemoryPool;

int main(int argc, char** argv)
{
    MemoryPool pool(1024);          // unlimited number of 1024 byte blocks
//  MemoryPool pool(1024, 4, 16); // at most 16 blocks; 4 preallocated

    char* buffer = reinterpret_cast<char*>(pool.get());

    std::cin.read(buffer, pool.blockSize());

    std::streamsize n = std::cin.gcount();
    std::string s(buffer, n);

    pool.release(buffer);

    std::cout << s << std::endl;

    return 0;
}
```

Singletons

> [...] The singleton design pattern is used to restrict instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system. Sometimes it is generalized to systems that operate more efficiently when only one or a few objects exist. It is also considered an anti-pattern since it is often used as a euphemism for global variable. Before designing a class as a singleton, it is wise to consider whether it would be enough to design a normal class and just use one object.
>
> Wikipedia

Singletons

- If you *absolutely* must have a singleton...
- POCO provides a Poco::SingletonHolder class that helps with the thread-safe management of lazy-created singletons.
- #include "Poco/SingletonHolder.h"
- The singleton instance is created on the heap when it is requested for the first time.
- The singleton instance is destroyed when the application terminates.

Singletons and DCLP

- What about DCLP?
- Glad you asked...
- Repeat after me...
  - DCLP is broken. I must not use it.
- Want proof?
  - http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf
  - http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html
  - http://en.wikipedia.org/wiki/Double-checked_locking

```
#include "Poco/SingletonHolder.h"

class MySingleton
{
public:
    MySingleton()
    {
        // ...
    }

    ~MySingleton()
    {
        // ...
    }

    // ...

    static MySingleton& instance()
    {
        static Poco::SingletonHolder<MySingleton> sh;
        return *sh.get();
    }
};
```

applied informatics

www.appinf.com | info@appinf.com
T +43 4253 32596 | F +43 4253 32096