auto\_ptr

**auto\_ptr** type is provided by the C++ standard library as a sort of smart pointer that helps to avoid resource leaks when exceptions are thrown.

Here is a typical example which has potential of memory leak.

void memory\_leak()

{

ClassA \* ptr = new ClassA;

...

delete ptr;

}

The reason why this function is source of trouble is that the deletion of the object might be forgotten especially if we have **return** inside of it. Also an exception would exit the function before the **delete** statement at the end of the function causing a resource leak.

Usually, we do try to capture all exceptions as in the example below.

void memory\_leak()

{

ClassA \* ptr = new ClassA;

try {

...

}

catch(...) {

delete ptr;

throw;

}

delete ptr;

}

As we see in the example, trying to handle the deletion of this object properly in the event of an exception makes the code more complicated and redundant.

So, we need a pointer which can free the data to which it points whenever the pointer itself gets destroyed. Because the pointer is a local variable, it will be destroyed automatically when the function is exited regardless of whether the exit is normal or caused by an exception.

In other words, if an exception occurs after successful memory allocation but before the delete statement executes, a memory leak could occur. The C++ standard provides class template **auto\_ptr** in header file **<memory>** to deal with this situation.

Our **auto\_ptr** is a pointer that serves as **owner** of the object to which it refers. So, an object gets destroyed automatically when its **auto\_ptr** gets destroyed.

The function in the 1st example can be rewritten using **auto\_ptr**

#include <memory>

void memory\_leak()

{

std::auto\_ptr<ClassA> ptr(new ClassA);

...

}

The **delete** statement and **catch** clause are no longer needed.

An **auto\_ptr** has the same interface as an ordinary pointer. Operator \* dereferences the object and operator -> provides access to a member if the object is a class or a structure.

**But the pointer arithmetic such as ++ is not defined.**

One more thing we should be careful about the usage of the pointer is that **auto\_ptr** does not allow us to initialize an object with an ordinary pointer by using the assignment syntax. So, we must initialize the **auto\_ptr** directly by using **its value**.

std::auto\_ptr<ClassA> ptr1(new ClassA); // RIGHT

std::auto\_ptr<ClassA> ptr1 = new ClassA; **// WRONG**

Here is the example of **auto\_ptr** in action:

#include <iostream>

#include <memory>

using namespace std;

class Double

{

public:

Double(double d = 0) : dValue(d) { cout << "constructor: " << dValue << endl; }

~Double() { cout << "destructor: " << dValue << endl; }

void setDouble(double d) { dValue = d; }

private:

double dValue;

};

int main()

{

auto\_ptr<Double> ptr(new Double(3.14));

(\*ptr).setDouble(6.28);

return 0;

}

The example creates **auto\_ptr** object **ptr** and initializes it with a pointer to a dynamically allocated **Double** object.

Because **ptr** is a local automatic variable in **main()**, **ptr** is destroyed when main terminates. The auto\_ptr destructor forces a delete of the Double object pointed to by **ptr**, which in turn calls the Double class destructor. The memory that Double occupies is released. The Double object will be deleted automatically when the auto\_ptr object's destructor gets called.

Only **one** auto\_ptr at a time can **own** a dynamically allocated object. Thus, the object cannot be an array. By using its overloaded assignment operator or copy constructor, an auto\_ptr can transfer ownership of the dynamic memory it manages. The last auto\_ptr object that maintains the pointer to the dynamic memory will delete the memory. This makes auto\_ptr an ideal mechanism for returning dynamically allocated memory to client code. When the auto\_ptr goes out of scope in the client code, the auto\_ptr's destructor deletes the dynamic memory.

Though **std::auto\_ptr** is responsible for managing dynamically allocated memory and automatically calls delete to free the dynamic memory when the auto\_ptr is destroyed or goes out of scope, auto\_ptr have some limitations.

1. An auto\_ptr can't point to an array. When deleting a pointer to an array we must use **delete[]** to ensure that destructors are called for all objects in the array, but auto\_ptr uses **delete**.
2. It can't be used with the STL containers-elements in an STL container. When an auto\_ptr is copied, ownership of the memory is transferred to the new auto\_ptr and the original is set to NULL. In other words, auto\_ptrs don't work in STL containers because the containers, or algorithms manipulating them, might copy the stored elements. Copies of auto\_ptrs aren't equal because the original is set to NULL after being copied. An STL container may make copies of its elements, so you can't guarantee that a valid copy of the auto\_ptr will remain after the algorithm processing the container's elements finishes.

An **auto\_ptr** is simply an object that holds a pointer for us within a function. Holding a pointer to guarantee deletion at the end of a scope is what **auto\_ptr** is for, and for other uses requires very specialized skills from a programmer.

The **Boost.Smart\_ptr library** provides additional smart pointers to fill in the gaps where auto\_ptrs don't work. TR1 includes two of the six types of smart pointers in the Boost.Smart\_ptr library, namely **shared\_ptr** and **weak\_ptr**. These smart pointers are not meant to replace auto\_ptr. Instead, they provide additional options with different functionality.

Smart Pointers

**TR1** specifies new components and all are in the **boost** namespace. For example, the full name of the **shared\_ptr** is:

boost::shared\_ptr

Here is the list of the new components:

smart pointers

A **smart pointer** is an **object that acts like a pointer** for most intents and purposes but avoids most of the problems inherent with C++ pointers. At its simplest, a smart pointer contains a native **pointer** as a data member and provides a set of **overloaded operators** that make it act like a pointer in most ways. Pointers can be dereferenced, so the \* and -> operators are overloaded to return the address as expected. Pointers can undergo pointer arithmetic operations, so the +, -, ++, and -- operators are also overloaded appropriately.

Because a smart pointer is an **object**, it can contain additional meta-data and take additional steps not possible with a regular pointer. For example, a smart pointer might contain information that allows it to recognize when the object to which it points has been deleted and start returning a NUll if so.

Smart pointers can also help with object lifetime management by cooperating with one another to determine the number of references to a particular object. This is called **reference counting**. When the number of smart pointers that reference a particular object drops to zero, we know that the object is no longer needed, so it can be automatically deleted. This can free the programmer from having to worry about object ownership and orphaned object (an object that still occupied memory but is no longer needed or referenced by any other object in the system).

Smart pointers have their share of problems. For one thing, they are relatively easy to implement, but they are extremely tough to get right. There are a great many cases to handle, and the [std::auto\_ptr](http://www.bogotobogo.com/cplusplus/autoptr.php" \t "_blank) class provided by the standard C++ library is widely recognized to be inadequate in many situations, and it's now deprecated.

1. **shared\_ptr**

The **auto\_ptr** has unusual characters: copying it whether via copy constructor or copy assignment operator sets it to null, and the copying pointer assumes ownership of the resource as we see in the example below:

#include <iostream>

#include <memory>

using namespace std;

class A{};

int main()

{

auto\_ptr<A> pA(new A);

cout << pA.get() << endl;

auto\_ptr<A> pB(pA);

cout << pA.get() << endl;

cout << pB.get() << endl;

return 0;

}

Output is:

001B0950

00000000

001B0950

In the example, the **get()** method returns a pointer to the object pointed by the auto\_ptr object, if any, or zero if it does not point to any object.

Note that the second output is **null**. So, in the copy constructor, **pA** transferred the ownership of **A** object to **pB**.

This behavior and the underlying requirement that resources managed by **auto\_ptr**s must never have more than one **auto\_ptr** pointing to them, means that **auto\_ptr**s aren't the best way to handle resources which are dynamically allocated.

So, as an alternative to **auto\_ptr**, we have a **referencing-counting smart pointer**. It keeps track of how many objects point to a particular resource and deletes the resource automatically when nothing is pointing to it.

By replacing **auto\_ptr** with **share\_ptr**, with an almost same code below, it produces the output we want to:

#include <boost/smart\_ptr/shared\_ptr.hpp>

#include <iostream>

#include <memory>

class A{};

int main()

{

boost::shared\_ptr<A> pA(new A);

std::cout << pA.get() << std::endl;

boost::shared\_ptr<A> pB(pA);

std::cout << pA.get() << std::endl;

std::cout << pB.get() << std::endl;

return 0;

}

Output is:

002C0950

002C0950

002C0950

Since copying **boost::shared\_ptr** works as we expect, it can be used in **STL** containers while we cannot use **std::auto\_ptr** for STL containers.

The major problem being solved using **share\_ptr** is to know the correct time to delete a resource that is shared. The following example has two classes, **A** and **B**. The classes are sharing an instance of **int**, and store a **shared\_ptr<int>**. When we create the instances of each class, the **shared\_ptr pTemp** is passed to the constructors. In other words, all three **shared\_ptr**s, are now referring to the same instance of an **int**. If we had used pointers to achieve such sharing of an **int**, each class would have had a hard time figuring out when it should be deleted. In the example, until the end of **main()**, the reference count is **3**. If all of the pointers go out of scope, the reference count reaches **0**, allowing the shared instance of **int** to be deleted.

**shared\_ptr** holds an internal pointer to a resource such as a dynamically allocated object that may be shared with other objects in the program. We can have any number of**shared\_ptr**s to the same resource. **shared\_ptr** really does share the resource, if we change the resource with one shared\_ptr, the changes also will be seen by the other**shared\_ptr**s. The internal pointer is deleted once the last **shared\_ptr** to the resource is destroyed. **shared\_ptr** uses **reference counting** to determine how many **shared\_ptr**s point to the resource. Each time a new **shared\_ptr** to the resource is created, the reference count increases, and each time one is destroyed, the reference count decreases. When the reference count reaches zero, the internal pointer is deleted and the memory is released.

#include <boost/smart\_ptr/shared\_ptr.hpp>;

#include <iostream>

#include <memory>

class classA

{

boost::shared\_ptr<int> ptA;

public:

classA(boost::shared\_ptr<int> p) : ptA(p) {}

void setValue(int n) {

\*ptA = n;

}

};

class classB

{

boost::shared\_ptr<int> ptB;

public:

classB(boost::shared\_ptr<int> p) : ptB(p) {}

int getValue() const {

return \*ptB;

}

};

int main()

{

boost::shared\_ptr<int> pTemp(new int(2013));

classA a(pTemp);

classB b(pTemp);

a.setValue(2014);

std::cout << "b.getValue() = " << b.getValue() << std::endl;

return 0;

}

Output is:

b.getValue() = 2014

**shared\_ptr** also allows us to determine how the resource will be destroyed. For most dynamically allocated objects, **delete** is used. However, some resources require more complex cleanup. In that case, we can supply a custom deleter function, or function object, to the **shared\_ptr** destructor. The deleter determines how to destroy the resource. When the reference count reaches zero and the resource is ready to be destroyed, the **shared\_ptr** calls the **custom deleter function**. This functionality enables a **shared\_ptr** to manage almost any kind of resource.

For **dynamically allocated arrays**, we shouldn't use either of them because they use **delete** in their destructor but not **delete[]**. We can use **vector** instead. If we insist on using boost, we can use either **boost::shared\_array** or **boost::scoped\_array**.

1. **shared\_array**  
   A pointer to an array of objects whose lifetimes are shared by multiple owners.
2. **scoped\_ptr**

A pointer to a single object with one owner.   
Let's look at the following timer example used in [Design Patterns: Nested Implementation Class](http://www.bogotobogo.com/DesignPatterns/introduction.php#hiding_implementation_detail_nested_class)

// timer.h

class Timer

{

public:

explicit Timer(double);

~Timer();

private:

class Implementation;

Implementation \*pImpl;

};

// timer.cpp

#include "timer.h"

#include <iostream>

#ifdef WIN32

#include <Windows.h>

#else

#include <sys/time.h>

#endif

class Timer::Implementation

{

public:

double elapsedTime()

{

#ifdef WIN32

return (GetTickCount() - mStartTime)/1000 ;

#else

struct timeval endTime;

gettimeofday(&endTime;,NULL);

return endTime.tv\_sec+(endTime.tv\_usec/1000000.0)-

(mStartTime.tv\_sec+(mStartTime.tv\_usec/1000000.0)) ;

#endif

}

#ifdef WIN32

DWORD mStartTime;

#else

struct timeval mStartTime;

#endif

double mDuration;

};

Timer::Timer(double d):pImpl(new Timer::Implementation())

{

pImpl->mDuration = d;

#ifdef WIN32

pImpl->mStartTime = GetTickCount();

#else

gettimeofday(&(pImpl->mStartTime), NULL);

#endif

}

Timer::~Timer()

{

while(pImpl->elapsedTime() < pImpl->mDuration) ;

std::cout << pImpl->mDuration << " sec elapsed" << std::endl;

delete pImpl;

pImpl = NULL;

}

// main.cpp

#include "timer.h"

int main()

{

double wait = 5;

Timer \*pTimer = new Timer(wait);

delete pTimer;

return 0;

}

As we see in the code above, we allocate **Implementation** object inside the **Timer**constructor, and initialize the private member, **pImpl** of the class. This approach is error-prone, and every now an then we may forget to delete the object in our destructor.

Using smart pointers make easier. In other words, we could use a **shared** pointer or a **scoped** pointer to hold the **Implementation object pointer**.

Here is the new code which is using **scope\_ptr**:

// timer.h

#include <boost/smart\_ptr/scoped\_ptr.hpp>

class Timer

{

public:

explicit Timer(double);

~Timer();

private:

class Implementation;

//Implementation \*pImpl;

boost::scoped\_ptr<Implementation> pImpl;

};

// timer.cpp

#include "timer.h"

#include <iostream>

#ifdef WIN32

#include <Windows.h>

#else

#include <sys/time.h>

#endif

class Timer::Implementation

{

public:

double elapsedTime()

{

#ifdef WIN32

return (GetTickCount() - mStartTime)/1000 ;

#else

struct timeval endTime;

gettimeofday(&endTime;,NULL);

return endTime.tv\_sec+(endTime.tv\_usec/1000000.0)-

(mStartTime.tv\_sec+(mStartTime.tv\_usec/1000000.0)) ;

#endif

}

#ifdef WIN32

DWORD mStartTime;

#else

struct timeval mStartTime;

#endif

double mDuration;

};

Timer::Timer(double d):pImpl(new Timer::Implementation())

{

pImpl->mDuration = d;

#ifdef WIN32

pImpl->mStartTime = GetTickCount();

#else

gettimeofday(&(pImpl->mStartTime), NULL);

#endif

}

Timer::~Timer()

{

while(pImpl->elapsedTime() < pImpl->mDuration) ;

std::cout << pImpl->mDuration << " sec elapsed" << std::endl;

// delete pImpl;

// pImpl = NULL;

}

// main.cpp

#include "timer.h"

int main()

{

double wait = 5;

Timer \*pTimer = new Timer(wait);

delete pTimer;

return 0;

}

We could have used a **boost::shared\_ptr**, which would mean that any copy would point to the same **Implementation** object. In any case, using either a **shared\_ptr** or a **scoped\_ptr** means that the **Implementation** object will be freed automatically when the object is destroyed. So, we no longer need to delete it in the destructor as shown in commented lines of the new code, **Timer::~Timer()**.

1. **scoped\_array**

A pointer to an array of objects with one owner.

1. **weak\_ptr**  
   A pointer that does not own or automatically destroy the object it references (whose lifetime is assumed to be managed by a **shared\_ptr**). We could think it as a **shared\_ptr observer**.

A **weak\_ptr** points to the resource managed by a **shared\_ptr** without assuming any responsibility for it. The reference count for a **shared\_ptr** doesn't increase when a **weak\_ptr** references it. That means that the resource of a **shared\_ptr** can be deleted while there are still **weak\_ptr** pointing to it. When the last **shared\_ptr** is destroyed, the resource is deleted and any remaining **weak\_ptr** are set to NULL. One use for **weak\_ptr**, as shown in the example below, is to avoid memory leaks caused by **circular references**.

A **weak\_ptr** can't directly access the resource it points to, we must create a shared\_ptr from the **weak\_ptr** to access the resource. There are two ways to do this.

* 1. We can pass the **weak\_ptr** to the **shared\_ptr** constructor. That creates a **shared\_ptr** to the resource being pointed to by the **weak\_ptr** and properly increases the reference count. If the resource has already been deleted, the **shared\_ptr** constructor will throw a **boost::bad\_weak\_ptr** exception.
  2. We can also call the **weak\_ptr** member function **lock()**, which returns a **shared\_ptr** to the **weak\_ptr**'s resource. If the **weak\_ptr** points to a deleted resource, lock will return an empty **shared\_ptr**. The **lock()** should be used when an empty **shared\_ptr** isn't considered an error. We can access the resource once you have a **shared\_ptr** to it. This approach is shown in the example below.

**weak\_ptr** should be used in any situation where we need to observe the resource but don't want to assume any management responsibilities for it. The example shows the use of **weak\_ptr**s in circularly referential data, a situation in which two objects refer to each other internally.

In the example below, we define classes **Singer** and **Song**. Each class has a pointer to an instance of the other class. This creates a **circular reference** between the two classes. Note that we use both **weak\_ptr** and **shared\_ptr** to hold the cross reference to each class.

Classes **Singer** and **Song** define destructors that each display a message to indicate when an instance of either class is destroyed. Each class also defines a member function to print the title of the Song and Singer's name. Because we can't access the resource directly through a **weak\_ptr**, first we create a **shared\_ptr** from the **weak\_ptr** data member **lock()**. If the resource the **weak\_ptr** is referencing doesn't exist, the call to the **lock()** function returns a **shared\_ptr** which points to NULL and the condition fails. Otherwise, the new **shared\_ptr** contains a valid pointer to the **weak\_ptr**'s resource, and we can access the resource. If the condition is true (when both **songPtr** and **singerPtr**aren't NULL), we print the reference count to show that it increased with the creation of the new **shared\_ptr**, then we print the title of the Song and Singer's name. The **shared\_ptr** is destroyed when the function exits so the reference count decreases by one.

Here are the codes:

**Singer.h**

#ifndef SINGER\_H

#define SINGER\_H

#include <string>

using std::string;

#include "boost/shared\_ptr.hpp"

#include "boost/weak\_ptr.hpp"

class Song;

class Singer

{

public:

Singer::Singer(const string &SingerName;);

~Singer();

void printSongTitle() ;

string name;

boost::weak\_ptr<Song> weakSongPtr;

boost::shared\_ptr<Song> sharedSongPtr;

};

#endif

**Song.h**

#ifndef SONG\_H

#define SONG\_H

#include <string>

using std::string;

#include "boost/shared\_ptr.hpp"

#include "boost/weak\_ptr.hpp"

class Singer; // forward declaration

class Song

{

public:

Song(const string &SongTitle;) ;

~Song();

void printSingerName();

string title;

boost::weak\_ptr<Singer> weakSingerPtr;

boost::shared\_ptr<Singer> sharedSingerPtr;

};

#endif

**Singer.cpp**

#include <string>

using namespace std;

#include "boost/shared\_ptr.hpp"

#include "boost/weak\_ptr.hpp"

#include "Song.h"

#include "Singer.h"

Singer::Singer(const string &SingerName;) : name(SingerName)

{

cout << "Singer constructor: " << name << endl;

}

Singer::~Singer()

{

cout << "Singer destructor: " << name << endl;

}

void Singer::printSongTitle()

{

// if weaksongPtr.lock() returns a non-empty shared\_ptr

if (boost::shared\_ptr<Song> songPtr = weakSongPtr.lock()) {

cout << "Reference count for song " << songPtr->title

<< " is " << songPtr.use\_count() << "." << endl;

cout << "Singer " << name << " wrote the song " << songPtr->title << "\n\n";

}

else // weaksongPtr points to NULL

cout << "This Singer has no song." << endl;

}

**Song.cpp**

#include <string>

using namespace std;

#include "boost/shared\_ptr.hpp"

#include "boost/weak\_ptr.hpp"

#include "Song.h"

#include "Singer.h"

Song::Song(const string &SongTitle;) : title(SongTitle)

{

cout << "Song constructor: " << title << endl;

}

Song::~Song()

{

cout << "Song destructor: " << title << endl;

}

void Song::printSingerName()

{

// if weakSingerPtr.lock() returns a non-empty shared\_ptr

if (boost::shared\_ptr<Singer> singerPtr = weakSingerPtr.lock() ) {

// show the reference count increase and print the Singer's name

cout << "Reference count for Singer " << singerPtr->name

<< " is " << singerPtr.use\_count() << "." << endl;

cout << "The Song " << title << " was written by "

<< singerPtr->name << "\n" << endl;

}

else // weakSingerPtr points to NULL

cout << "This Song has no Singer." << endl;

}

**main.cpp**

#include <iostream>

#include <string>

#include "boost/shared\_ptr.hpp"

#include "boost/weak\_ptr.hpp"

#include "Singer.h"

#include "Song.h"

using namespace std;

int main()

{

cout << "Creating a Song and an Singer ..." << endl;

boost::shared\_ptr<Song> SongPtr( new Song( "The Boys" ) );

boost::shared\_ptr<Singer> SingerPtr(new Singer( "Girls Generation" ) );

cout << "\nReferencing the Song and Singer to each other..." << endl;

SongPtr->weakSingerPtr = SingerPtr;

SingerPtr->weakSongPtr = SongPtr;

cout << "\nSetting the shared\_ptr data members to create the memory leak..." << endl;

SongPtr->sharedSingerPtr = SingerPtr;

SingerPtr->sharedSongPtr = SongPtr;

cout << "Reference count for SongPtr and SingerPtr should be one, but ... " << endl;

cout << "Reference count for Song " << SongPtr->title << " is "

<< SongPtr.use\_count() << endl;

cout << "Reference count for Singer " << SingerPtr->name << " is "

<< SingerPtr.use\_count() << "\n" << endl;

cout << "\nAccess the Singer's name and the Song's title through "

<< "weak\_ptrs." << endl;

SongPtr->printSingerName();

SingerPtr->printSongTitle();

cout << "Reference count for each shared\_ptr shoulb be back to one:" << endl;

cout << "Reference count for Song " << SongPtr->title << " is "

<< SongPtr.use\_count() << endl;

cout << "Reference count for Singer " << SingerPtr->name << " is "

<< SingerPtr.use\_count() << "\n" << endl;

// the shared\_ptrs go out of scope, the Song and Singer are destroyed

cout << "The shared\_ptrs are going out of scope." << endl;

return 0;

}

In main(), we see the memory leak caused by the circular reference between classes Singer and Song. The lines:

boost::shared\_ptr<Song> SongPtr( new Song( "The Boys" ) );

boost::shared\_ptr<Singer> SingerPtr(new Singer( "Girls Generation" ) );

create **shared\_ptr**s to an instance of each class. The **weak\_ptr** data members are set as:

SongPtr->weakSingerPtr = SingerPtr;

SingerPtr->weakSongPtr = SongPtr;

The **shared\_ptr** data members for each class are set in the lines below.

SongPtr->sharedSingerPtr = SingerPtr;

SingerPtr->sharedSongPtr = SongPtr;

The instances of classes **Singer** and **Song** now reference each other. We then print the reference count for the **shared\_ptr** to show that each instance is referenced by two shared\_ptrs

cout << "Reference count for Song " << SongPtr->title << " is "

<< SongPtr.use\_count() << endl;

cout << "Reference count for Singer " << SingerPtr->name << " is "

<< SingerPtr.use\_count() << "\n" << endl;

the ones we create in the **main()** and the data member of each instance. Recall that **weak\_ptr** don't affect the reference count. Then we call each class's member function to print the information stored in the **weak\_ptr** data member:

SongPtr->printSingerName();

SingerPtr->printSongTitle();

The functions also display the fact that another **shared\_ptr** was created during the function call. Finally, we print the reference counts again to show that the additional shared\_ptrs created in the **printSingerName** and **printSongTitle** member functions are destroyed when the functions finish.

cout << "Reference count for Song " << SongPtr->title << " is "

<< SongPtr.use\_count() << endl;

cout << "Reference count for Singer " << SingerPtr->name << " is "

<< SingerPtr.use\_count() << "\n" << endl;

The output is:

Creating a Song and an Singer ...

Song constructor: The Boys

Singer constructor: Girls Generation

Referencing the Song and Singer to each other...

Setting the shared\_ptr data members to create the memory leak...

Reference count for SongPtr and SingerPtr should be one, but ...

Reference count for Song The Boys is 2

Reference count for Singer Girls Generation is 2

Access the Singer's name and the Song's title through weak\_ptrs.

Reference count for Singer Girls Generation is 3.

The Song The Boys was written by Girls Generation

Reference count for song The Boys is 3.

Singer Girls Generation wrote the song The Boys

Reference count for each shared\_ptr shoulb be back to one:

Reference count for Song The Boys is 2

Reference count for Singer Girls Generation is 2

The shared\_ptrs are going out of scope.

At the end of **main()**, the **shared\_ptr** to the instances of **Singer** and **Song** we created go out of scope and are destroyed. Notice that the output doesn't show the destructors for classes **Singer** and **Song**. The program has a memory leak, in other words, the instances of **Singer** and **Song** aren't destroyed because of the **shared\_ptr** data members. When **songPtr** is destroyed at the end of the **main()**, the reference count for the instance of class **Song** becomes one because the instance of **Singer** still has a **shared\_ptr** to the instance of **Song**, so it's not deleted. When **singerPtr** goes out of scope and is destroyed, the reference count for the instance of class **Singer** also becomes one because the instance of **Song** still has a **shared\_ptr** to the instance of **Singer**. Neither instance is deleted because the reference count for each is still one.

Now, comment out the following lines:

//SongPtr->sharedSingerPtr = SingerPtr;

//SingerPtr->sharedSongPtr = SongPtr;

This prevents the code from setting the **shared\_ptr** data members for classes **Singer** and **Song**. Then, we'll have a new output:

Creating a Song and an Singer ...

Song constructor: The Boys

Singer constructor: Girls Generation

Referencing the Song and Singer to each other...

Setting the shared\_ptr data members to create the memory leak...

Reference count for SongPtr and SingerPtr should be one, but ...

Reference count for Song The Boys is 1

Reference count for Singer Girls Generation is 1

Access the Singer's name and the Song's title through weak\_ptrs.

Reference count for Singer Girls Generation is 2.

The Song The Boys was written by Girls Generation

Reference count for song The Boys is 2.

Singer Girls Generation wrote the song The Boys

Reference count for each shared\_ptr shoulb be back to one:

Reference count for Song The Boys is 1

Reference count for Singer Girls Generation is 1

The shared\_ptrs are going out of scope.

Singer destructor: Girls Generation

Song destructor: The Boys

Notice that the initial reference count for each instance is now **one** instead of **two**because we don't set the **shared\_ptr** data members. The last two lines of the output show that the instances of classes **Singer** and **Song** were destroyed at the end of the **main()**. We eliminated the memory leak by using the **weak\_ptr** data members rather than the **shared\_ptr** data members. The **weak\_ptr** don't affect the reference count but still allow us to access the resource when we need it by creating a temporary **shared\_ptr**to the resource. When the **shared\_ptr** we created in **main()** are destroyed, the reference counts become **zero** and the instances of classes **Singer** and **Song** are deleted properly.

1. **intrusive\_ptr**

A pointer that implements reference counting by assuming that the pointed-to object will maintain the reference count itself. Intrusive pointers are useful because they are the same size as the native C++ pointer (no reference-counting mechanism is required), and because they can be constructed directly from native pointers.

Properly implementing a smart pointer class can be a daunting task, and all sorts of issues come up, including:

1. type safety of smart pointers
2. the ability for a smart pointer to be used with an incomplete type
3. correct smart pointer behavior when an exception occurs.
4. runtime costs, which can be high

Smart pointers - pros and cons

This is from [Google C++ Style Guide](http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Smart_Pointers).

Smart pointers are objects that act like pointers, but automate management of the underlying memory. If we actually need pointer semantics, **scoped\_ptr** is great. We should only use **std::tr1::shared\_ptr** with a non-const referent when it is truly necessary to share ownership of an object (e.g. inside an STL container). We should never use **auto\_ptr**.

1. **Pros**  
   Smart pointers are extremely useful for preventing **memory leaks**, and are essential for writing **exception-safe** code. They also formalize and document the **ownership** of dynamically allocated memory.
2. **Cons**  
   We prefer designs in which objects have single, fixed owners. Smart pointers which enable sharing or transfer of ownership can act as a tempting alternative to a careful design of ownership semantics, leading to confusing code and even bugs in which memory is never deleted. The semantics of smart pointers (especially **auto\_ptr**) can be nonobvious and confusing. The exception-safety benefits of smart pointers are not decisive, since we do not allow exceptions.
3. **Decision**  
   1. scoped\_ptr  
      Straightforward and risk-free. Use wherever appropriate.
   2. auto\_ptr  
      Confusing and bug-prone ownership-transfer semantics. Do not use.
   3. shared\_ptr  
      Safe with const referents (i.e. **shared\_ptr<const T>**). Reference-counted pointers with non-const referents can occasionally be the best design, but try to rewrite with single owners where possible.