**Smart Pointers**

During construction, it owns the memory and releases the same when it goes out of scope.

**auto\_ptr:**

**issue1 -** It has more fundamental flaws over its smartness. auto\_ptr transfers the ownership when it is assigned to another auto\_ptr.

eg. I have an auto\_ptr in Foo( ) and this pointer is passed another function say Fun( ) from Foo. Now once Fun( ) completes its execution, the ownership is not returned back to Foo.

**issue2**- auto\_ptr cannot be used with an array of objects. I mean it cannot be used with the operator new[].

**issue3** - auto\_ptr cannot be used with standard containers like vector, list, map, etc

**shared\_ptr:**

Multiple shared pointers can refer to a single object and when the last shared pointer goes out of scope, memory is released automatically.

* get( ) : To get the resource associated with the shared\_ptr.
* reset( ) : To yield the ownership of the associated memory block. If this is the last shared\_ptr owning the resource, then the resource is released automatically.
* unique: To know whether the resource is managed by only this shared\_ptr instance.
* operator bool: To check whether the shared\_ptr owns a memory block or not. Can be used with an if condition.

**issue1**- If a memory is block is associated with shared\_ptrs belonging to a different group, then there is an error.

eg.

int\* p = new int; shared\_ptr<int> sptr1( p); shared\_ptr<int> sptr2( p ); // error  
**issue2** - It may cause crash, if by mistake a programmer deletes the naked pointer p before the scope of the shared  
 pointer ends.  
  
**issue3 -** Cyclic Reference: Resources are not released properly if a cyclic reference of shared pointers are   
involved. Consider the following piece of code.  
  
**weak\_ptr :**  
﻿A weak pointer provides sharing semantics and not owning semantics. This means a weak pointer can share a resource  
held by a shared\_ptr﻿  
A weak pointer constructor takes a shared pointer as one of its parameters. Creating a weak pointer out of a   
shared pointer increases the *weak reference* counter of the shared pointer. This means that the shared pointer   
shares it resource with another pointer.   
if the strong reference of the shared pointer goes to 0, then the resource is released irrespective of the weak   
reference value.

1. Call the use\_count( ) method to know the count. Note that this method returns the strong reference count and not the weak reference.
2. Call the expired( ) method. This is faster than calling use\_count( ).

To get a shared\_ptr from a weak\_ptr call lock( ) or directly casting the weak\_ptr to shared\_ptr.

eg.

shared\_ptr<Test> sptr( new Test ); weak\_ptr<Test> wptr( sptr ); shared\_ptr<Test> sptr2 = wptr.lock( );  
  
Cyclic reference resolved using weak\_ptr  
eg.

class B;

class A

{

public:

A( ) : m\_a(5) { };

~A( )

{

cout<<" A is destroyed"<<endl;

}

void PrintSpB( );

weak\_ptr<B> m\_sptrB;

int m\_a;

};

class B

{

public:

B( ) : m\_b(10) { };

~B( )

{

cout<<" B is destroyed"<<endl;

}

weak\_ptr<A> m\_sptrA;

int m\_b;

};

void A::PrintSpB( )

{

if( !m\_sptrB.expired() )

{

cout<< m\_sptrB.lock( )->m\_b<<endl;

}

}

void main( )

{

shared\_ptr<B> sptrB( new B );

shared\_ptr<A> sptrA( new A );

sptrB->m\_sptrA = sptrA;

sptrA->m\_sptrB = sptrB;

sptrA->PrintSpB( );

}

**unique\_ptr:**

1. this is almost a kind of replacement to the error prone auto\_ptr.
2. unique\_ptr follows the exclusive ownership semantics, i.e., at any point of time, the resource is owned by only one unique\_ptr.
3. When unique\_ptr goes out of scope, the resource is released. If the resource is overwritten  
   by some other resource, the previously owned resource is released.
4. The unique\_ptr class provides the specialization to create an array of objects which calls delete[ ] instead of delete when the pointer goes out of scope.
5. Ownership of the resource can be transferred from one unique\_ptr to another by assigning it
6. unique\_ptr does not provide you copy semantics [copy assignment and copy construction is not possible] but move   
   semantics.