CS302 - Data Structures *using C++*

Topic: Safe Memory Management using Smart Pointers

Kostas Alexis

Raw Pointers

- Allocate memory in free store by using the new operator
  - Returns reference to newly created object in memory
- Store reference to object in a pointer variable
  - Use pointer variable to access object
- Copy reference to another pointer variable
  - Creates alias to same object

Raw Pointers

- Use delete operator to deallocate object's memory
    - Must also set to nullptr any pointer variables that referenced the object
- Need to keep track number of aliases that reference an object … else results in
    - Dangling pointers
    - Memory leaks
    - Other errors (program crash, wasted memory, …)

Raw Pointers

- Languages such as Java and Python disallow direct reference to objects
  - Use reference counting to track number of aliases that reference an object
  - Known as the “reference count”
- Language can detect when object no longer has references
  - Dangling pointers
  - Memory leaks
  - Other errors (program crash, wasted memory, …)

Smart Pointers

- C++ now supports “smart” pointers (or managed pointers)
  - Act like raw pointers
  - Also provide automatic memory management

Smart Pointers

- C++ now supports “smart” pointers (or managed pointers)
  - Act like raw pointers
  - Also provide automatic memory management
- When you declare a smart pointer
  - Placed on application stack
  - Smart pointer references an object – object is “managed”

Smart Pointers

- Smart-pointer templates
  - **shared_ptr** – provides shared ownership of an object

Smart Pointers

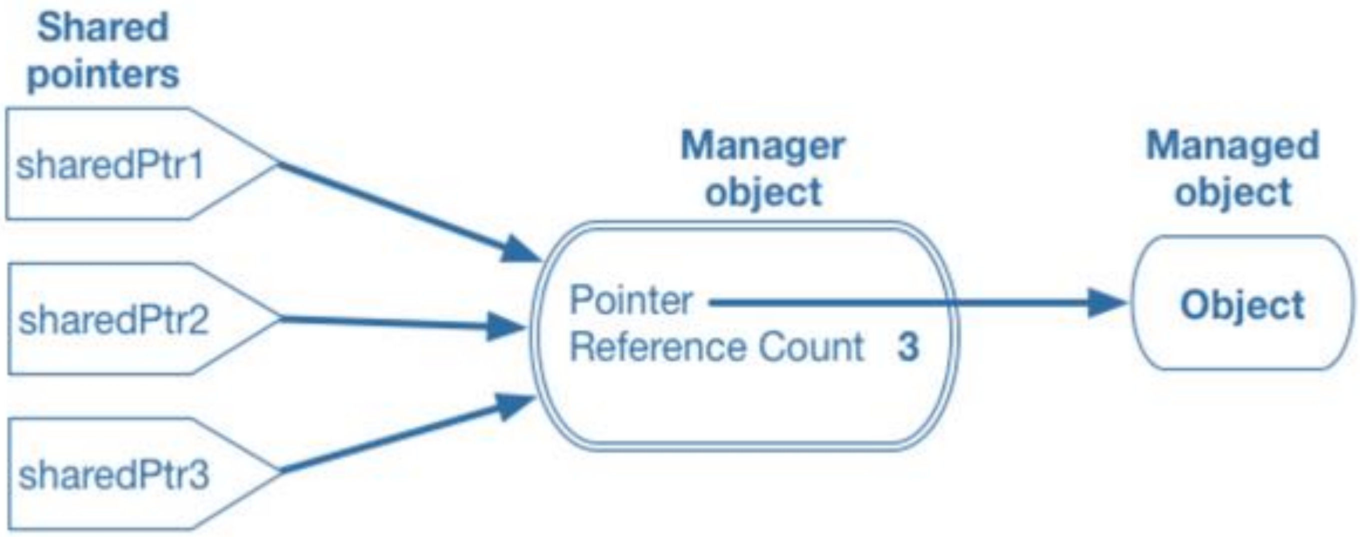
- Smart-pointer templates
  - **shared_ptr** – provides shared ownership of an object
  - **unique_ptr** – no other point can reference same object

Smart Pointers

- Smart-pointer templates
  - **shared_ptr** – provides shared ownership of an object
  - **unique_ptr** – no other point can reference same object
  - **weak_ptr** – reference to an object already managed by a shared pointer / it does not have ownership of the object

Using Shared Pointers

- Shared pointers – manager object referencing a managed object



Using Shared Pointers

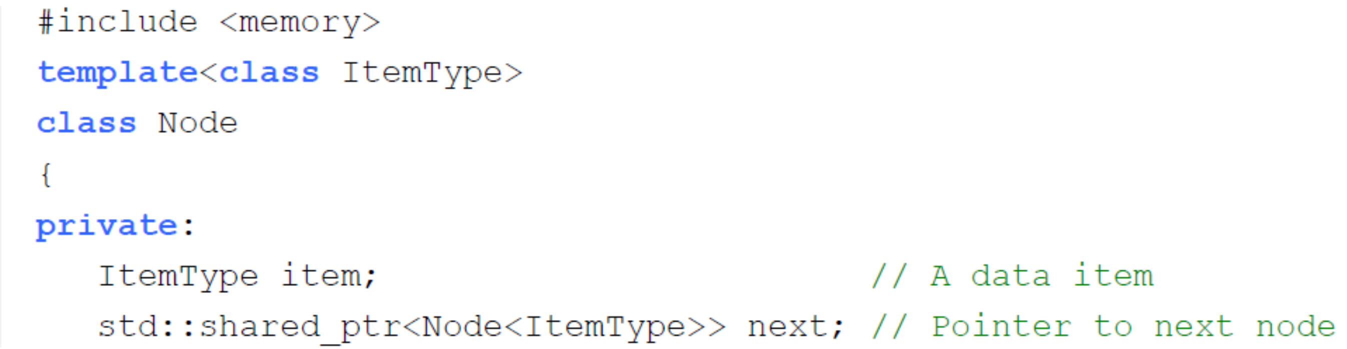
- A shared pointer
  - Provides a safe mechanism to implement shared object ownership
  - Maintains a count of aliases to an object
  - Decreases or increases the reference count of a managed object every time an instance is created or goes out of scope, or is assigned nullptr
  - Calls destructor of a managed object when reference count reaches 0

Revised Node and LinkedList Classes

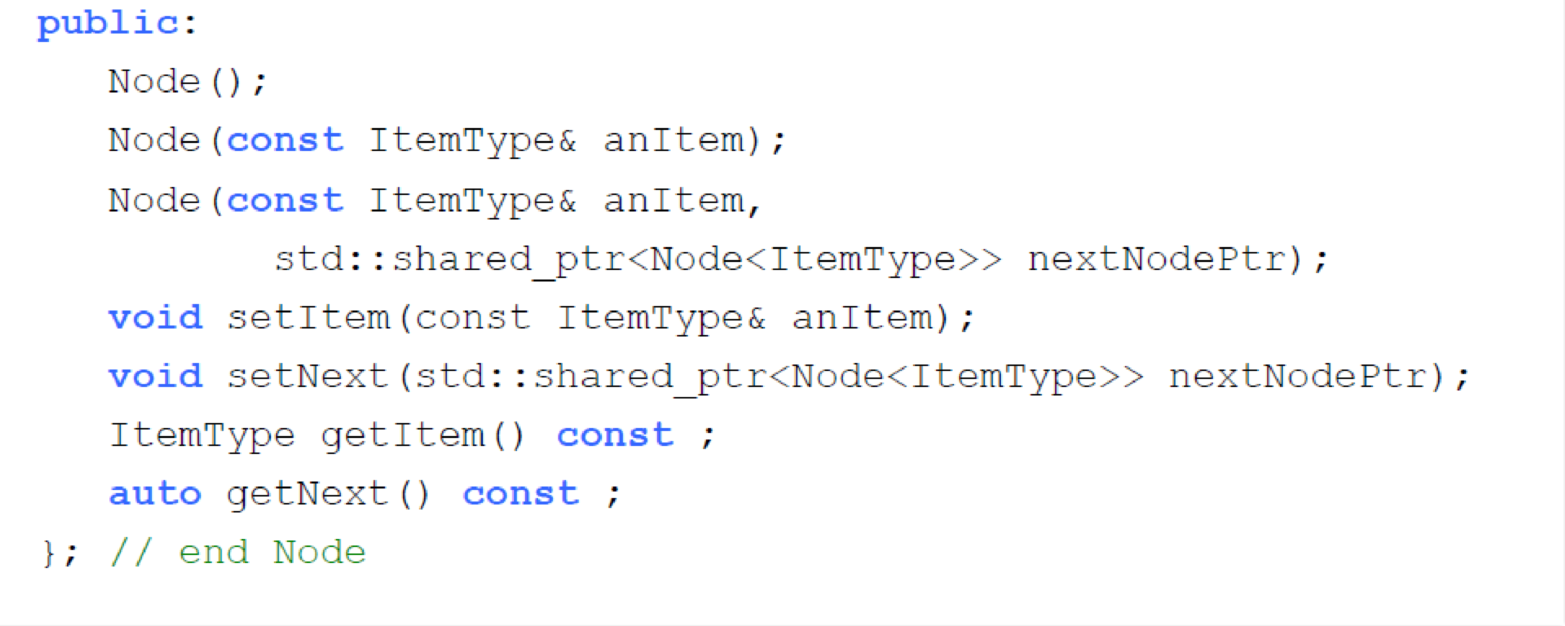
- Goal: Use shared pointers in previously described Node and LinkedList classes
    - Help ensure memory is handled correctly.

Revised Node and LinkedList Classes

- The revised header file for the class Node



```
#include <memory>
template<class ItemType>
class Node
{
private:
   ItemType item;                          // A data item
   std::shared_ptr<Node<ItemType>> next; // Pointer to next node
```



```
public:
   Node();
   Node(const ItemType& anItem);
   Node(const ItemType& anItem,
          std::shared_ptr<Node<ItemType>> nextNodePtr);
   void setItem(const ItemType& anItem);
   void setNext(std::shared_ptr<Node<ItemType>> nextNodePtr);
   ItemType getItem() const ;
   auto getNext() const ;
}; // end Node
```

Revised Node and LinkedList Classes

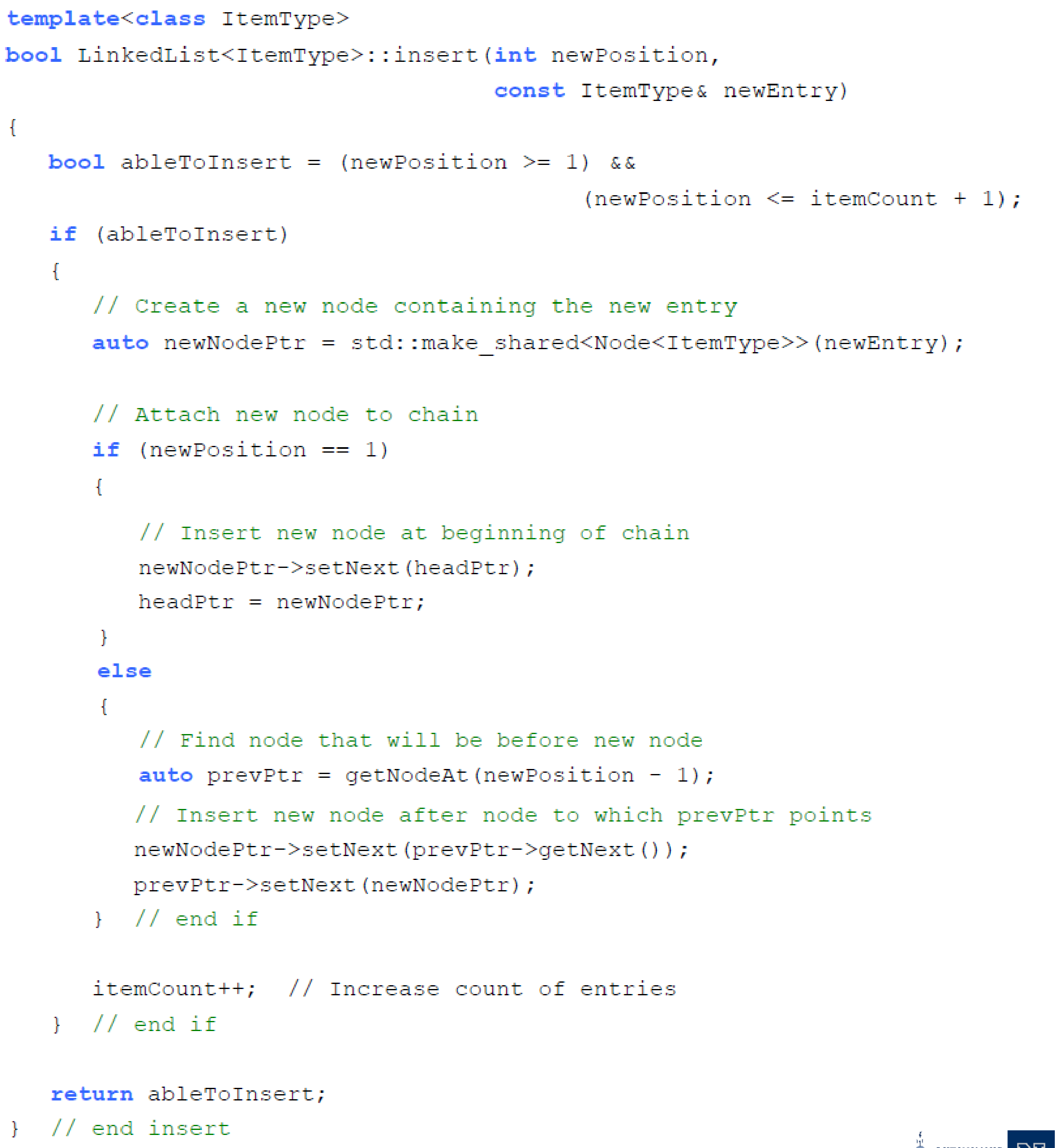
- The revised implementation file for the class Node

```
#include "Node.h"

template<class ItemType>
Node<ItemType>::Node()
{ } // end default constructor

template<class ItemType>
Node<ItemType>::Node(const ItemType& anItem)
                  : item(anItem)
{ } // end constructor

template<class ItemType>
Node<ItemType>::Node(const ItemType& anItem,
                     std::shared_ptr<xNode<ItemType>> nextNodePtr)
                  : item(anItem), next(nextNodePtr)
{ } // end constructor
```

```
template<class ItemType>
void Node<ItemType>::setItem(const ItemType& anItem)
{
   item = anItem;
} // end setItem

template<class ItemType>
void Node<ItemType>::setNext(std::shared_ptr<Node<ItemType>> nextNodePtr)
{
   next = nextNodePtr;
} // end setNext

template<class ItemType>
ItemType Node<ItemType>::getItem() const
{
   return item;
} // end getItem
template<class ItemType>
auto Node<ItemType>::getNext() const
{
   return next;
} // end getNext
```

Revised Node and

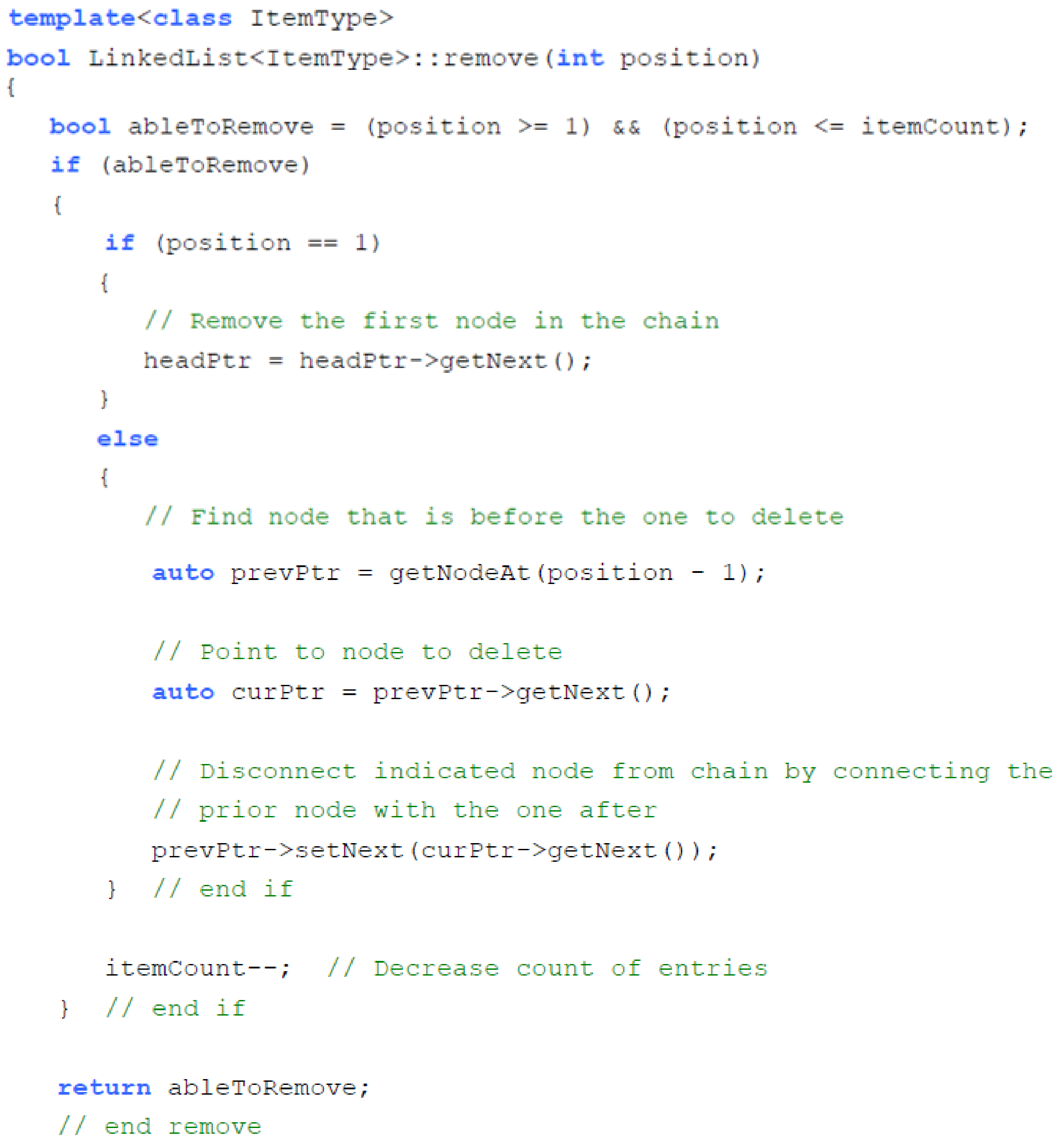
- The insert method for LinkedList



```
template<class ItemType>
bool LinkedList<ItemType>::insert(int newPosition,
                                  const ItemType& newEntry)
{
   bool ableToInsert = (newPosition >= 1) &&
                       (newPosition <= itemCount + 1);
   if (ableToInsert)
   {
      // Create a new node containing the new entry
      auto newNodePtr = std::make_shared<Node<ItemType>>(newEntry);

      // Attach new node to chain
      if (newPosition == 1)
      {
         // Insert new node at beginning of chain
         newNodePtr->setNext(headPtr);
         headPtr = newNodePtr;
      }
      else
      {
         // Find node that will be before new node
         auto prevPtr = getNodeAt(newPosition - 1);
         // Insert new node after node to which prevPtr points
         newNodePtr->setNext(prevPtr->getNext());
         prevPtr->setNext(newNodePtr);
      }  // end if

      itemCount++;  // Increase count of entries
   }  // end if

   return ableToInsert;
}  // end insert
```

Revised Node and L

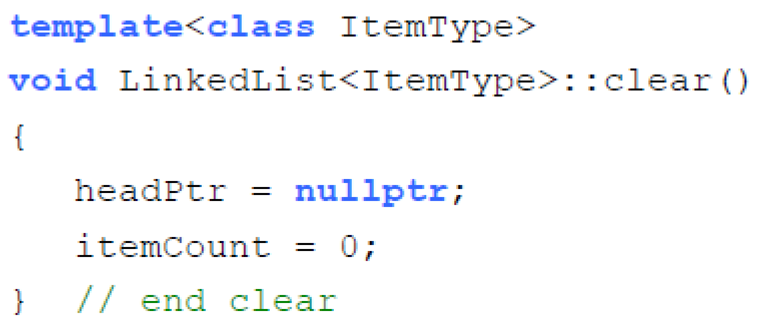
- The remove method for LinkedList



```
template<class ItemType>
bool LinkedList<ItemType>::remove(int position)
{
   bool ableToRemove = (position >= 1) && (position <= itemCount);
   if (ableToRemove)
   {
      if (position == 1)
      {
         // Remove the first node in the chain
         headPtr = headPtr->getNext();
      }
      else
      {
         // Find node that is before the one to delete
         auto prevPtr = getNodeAt(position - 1);

         // Point to node to delete
         auto curPtr = prevPtr->getNext();

         // Disconnect indicated node from chain by connecting the
         // prior node with the one after
         prevPtr->setNext(curPtr->getNext());
      }  // end if

      itemCount--;  // Decrease count of entries
   }  // end if

   return ableToRemove;
   // end remove
```

Revised Node and LinkedList Classes

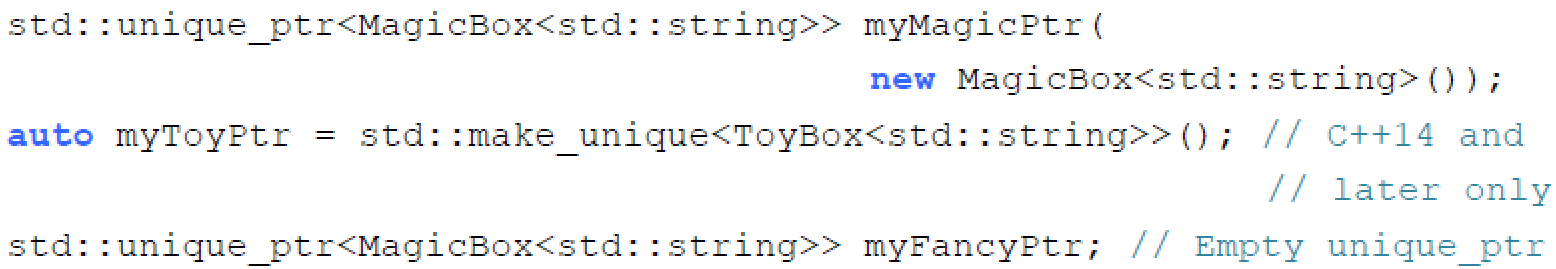
- The clear method for LinkedList



```
template<class ItemType>
void LinkedList<ItemType>::clear()
{
   headPtr = nullptr;
   itemCount = 0;
}  // end clear
```

Using Unique Pointers

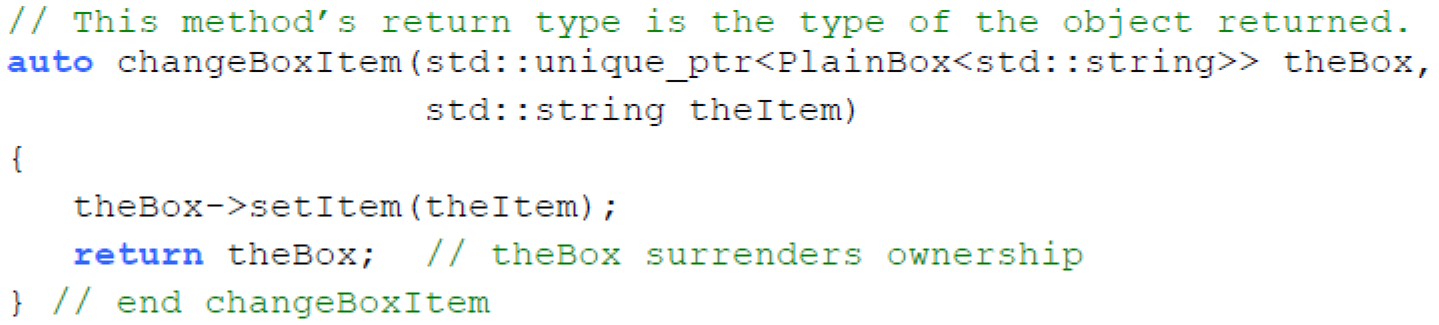
- Different ways to create unique pointers



```
std::unique_ptr<MagicBox<std::string>> myMagicPtr(
                                       new MagicBox<std::string>());
auto myToyPtr = std::make_unique<ToyBox<std::string>>(); // C++14 and
                                                         // later only
std::unique_ptr<MagicBox<std::string>> myFancyPtr; // Empty unique_ptr
```

Using Unique Pointers

- Function that accepts ownership of an object and then returns it to the caller



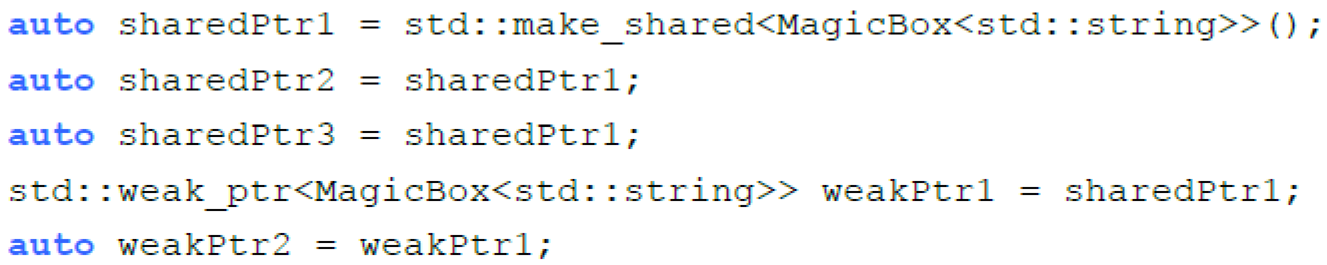
```
// This method’s return type is the type of the object returned.
auto changeBoxItem(std::unique_ptr<PlainBox<std::string>> theBox,
                   std::string theItem)
{
   theBox->setItem(theItem);
   return theBox;  // theBox surrenders ownership
} // end changeBoxItem
```

Using Unique Pointers

- A unique pointer
  - Has solitary ownership of its managed object
  - Behaves as if it maintains a reference count of either 0 or 1 for its managed object
  - Can transfer its unique ownership of its managed object to another unique pointer using method move
  - Cannot be assigned to another unique pointer

Using Weak Pointers

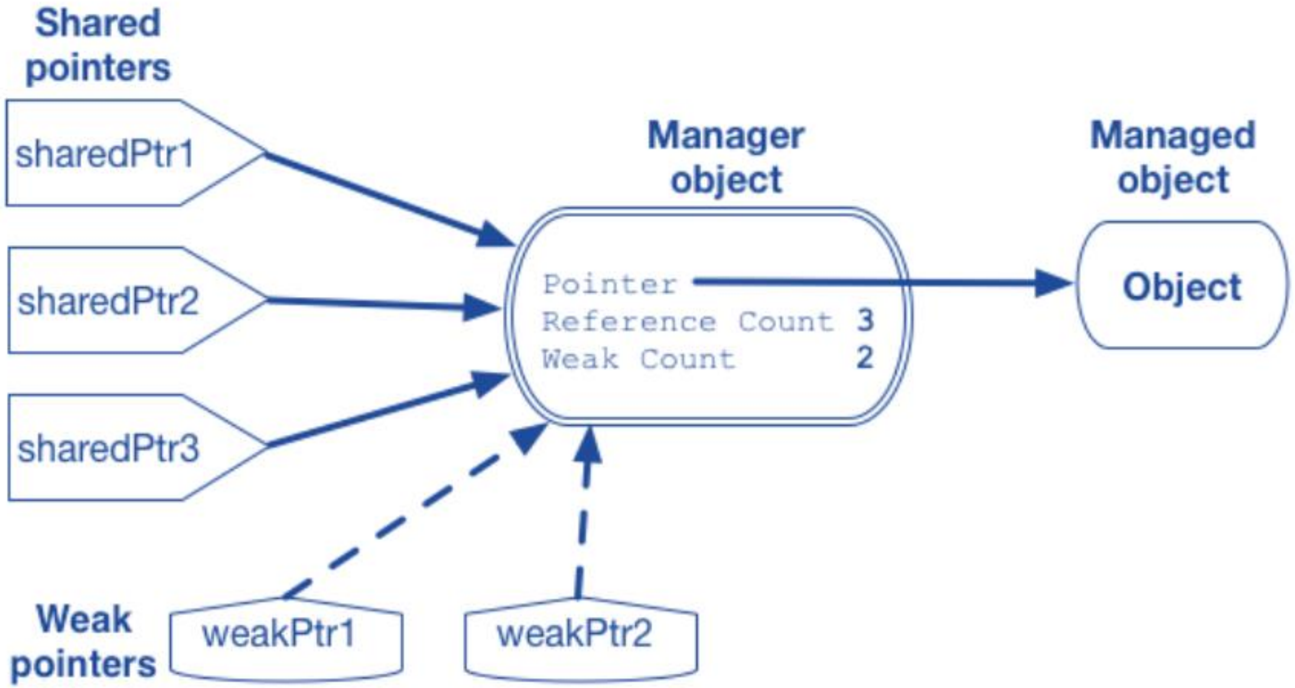
- Weak pointer only observes managed object
  - But does not have ownership
  - Therefore, cannot affect its lifetime
- After these statements execute, reference count for object managed by sharedPtr1 is 3



```cpp
auto sharedPtr1 = std::make_shared<MagicBox<std::string>>();
auto sharedPtr2 = sharedPtr1;
auto sharedPtr3 = sharedPtr1;
std::weak_ptr<MagicBox<std::string>> weakPtr1 = sharedPtr1;
auto weakPtr2 = weakPtr1;
```

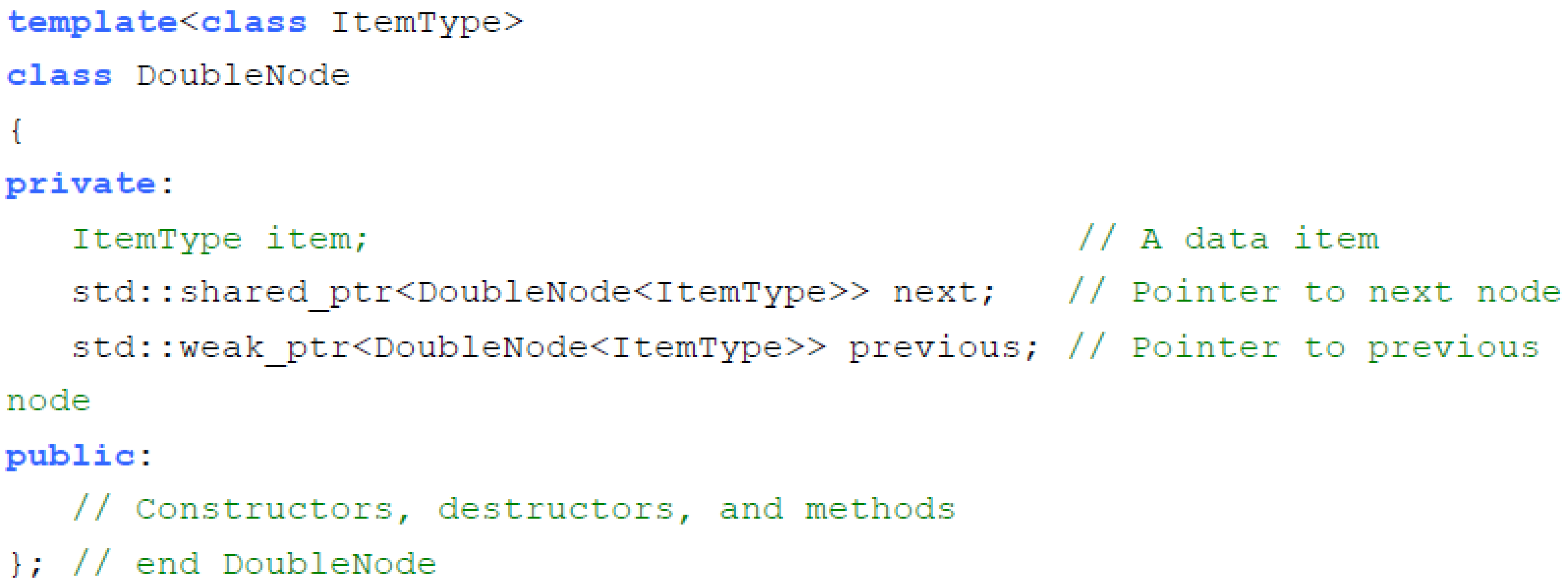
Using Weak Pointers

- Weak and shared ownership of a managed object



Using Weak Pointers

- Partial header file for the class DoubleNode



```
template<class ItemType>
class DoubleNode
{
private:
   ItemType item;                                      // A data item
   std::shared_ptr<DoubleNode<ItemType>> next;    // Pointer to next node
   std::weak_ptr<DoubleNode<ItemType>> previous; // Pointer to previous
node
public:
   // Constructors, destructors, and methods
}; // end DoubleNode
```

Using Weak Pointers

- A weak pointer
  - References but does not own an object referenced by shared pointer
  - Cannot affect the lifetime of managed object
  - Does not affect reference count of managed object
  - Has method lock to provide a shared-pointer version of its reference
  - Has method expired to detect whether its reference object no longer exists.

Other Smart Pointer Features

- Method common to all smart pointers
  - reset
- Method common to all shared and unique pointers
  - get
- Methods exclusive to shared pointers
  - unique
  - use_count
- Methods exclusive to unique pointers
  - release
- Unique pointers with arrays
  - Use a unique pointer to manage a dynamic array

Thank you