Smart Pointers – C++11

C++ has many benefits of pointers but also several pitfalls if memory management is not performed correctly. Dangling pointers or memory leaks can result in errors that are difficult to find. C++11 includes a new class called `shared_ptr` that simplifies memory management and sharing of objects in memory.

The `shared_ptr` class is a template that is a wrapper around an object allocated from the heap. The wrapper uses **reference counting** to track how many other pointers reference the object. The counter starts at zero. The counter is incremented by one every time a new variables references the object. Similarly, the counter is decremented by one every time a variable ceases to reference the object (e.g., it is deleted or reassigned). If the counter reaches zero then the object can be safely deleted and the allocated memory returned to the heap. This is all performed automatically, which frees the programmer from having to write his or her own memory management code!

As an example, consider the following code which implements a simple linked list of the `Node` class. The class simply stores an integer. The code is written using the "old" format of linking classes via pointer and does not explicitly free the memory that is allocated in the `listTest` function. This means that the program has a memory leak when execution returns to the `main` function. This could cause memory problems if the program did not immediately exit.

```
// Linked list of a simple Node class using traditional pointers.
// Note that this version has a memory leak when execution returns to
// main.
#include <iostream>
using namespace std;

// A simple Node class. A full-featured class would have
// several more functions.
class Node
{
private:
	int num;
	Node *next;
public:
	Node();
	~Node();
	Node(int num, Node *nextPtr);
	int getNum();
	Node* getNext();
	void setNext(Node *nextPtr);
};

Node::Node() : num(0), next(nullptr)
{ }

Node::Node(int numVal, Node *nextPtr) : num(numVal), next(nextPtr)
{ }

Node::~Node()
{
```

```
        cout << "Deleting " << num << endl;
}

int Node::getNum()
{
        return num;
}

Node* Node::getNext()
{
        return next;
}

void Node::setNext(Node *nextPtr)
{
        next = nextPtr;
}

void listTest()
{
        // Create a linked list with 10->20->30
        Node *root = new Node(10, nullptr);
        root->setNext(new Node(20, nullptr));
        root->getNext()->setNext(new Node(30, nullptr));

        // Output the list
        Node *temp;
        temp = root;
        while (temp != nullptr)
        {
                cout << temp->getNum() << endl;
                temp = temp->getNext();
        }
}

int main()
{
        listTest();
        return 0;
}
```

Program output:

```
10
20
30
```

Note that despite the existence of a destructor for the `Node` class, the destructor is never called. This is because we never delete each node. The memory allocated in `listTest` is never freed so we have a memory leak in `main`. This is not really a problem since the program immediately exits (at which point memory is reclaimed) but if there were further processing after the call to `listTest` then we could have memory problems.

Next, consider the same program written with the `shared_ptr` class. We must include the `<memory>` library. Every occurrence of a pointer to the `Node` class is replaces with `shared_ptr<Node>` instead.

```
// Linked list of a simple Node class using smart pointers.
// There is no memory leak since ths shared_ptr class
// handles reference counting and memory deallocation.
#include <iostream>
#include <memory>
using namespace std;

// Class modified to use shared_ptr of Nodes.
class Node
{
private:
	int num;
	shared_ptr<Node> next;
public:
	Node();
	~Node();
	Node(int num, shared_ptr<Node> nextPtr);
	int getNum();
	shared_ptr<Node> getNext();
	void setNext(shared_ptr<Node> nextPtr);
};

Node::Node() : num(0), next(nullptr)
{ }

Node::~Node()
{
	cout << "Deleting " << num << endl;
}

Node::Node(int numVal, shared_ptr<Node> nextPtr) : num(numVal),
next(nextPtr)
{ }

int Node::getNum()
{
	return num;
}

shared_ptr<Node> Node::getNext()
{
	return next;
}

void Node::setNext(shared_ptr<Node> nextPtr)
{
	next = nextPtr;
}

void listTest()
```

```
{
	shared_ptr<Node> root(new Node(10, nullptr));
	shared_ptr<Node> next1(new Node(20, nullptr));
	shared_ptr<Node> next2;
	// After a shared_ptr is declared we can set it
	// using the reset function
	next2.reset(new Node(30, nullptr));
	// Link the nodes together
	root->setNext(next1);
	next1->setNext(next2);

	// Output the list
	shared_ptr<Node> temp;
	temp = root;
	while (temp != nullptr)
	{
		cout << temp->getNum() << endl;
		temp = temp->getNext();
	}
}

int main()
{
	listTest();
	cout << "Exiting program." << endl;
	return 0;
}
```

Program output:

```
10
20
30
Deleting 10
Deleting 20
Deleting 30
Exiting program.
```

Note that the linked list is automatically deallocated for us by the `shared_ptr` class when the variables go out of scope in the `listTest` function. This is done for us after the call to `listTest` exits, as indicated by the messages output by the `Node` destructor before the program exits.

As a further example, consider what would happen if there is a global variable that references the second item in the linked list. In this case the `shared_ptr` class will not delete the remainder of the items in the list when the `listTest` function exits. This is because the nodes are only deleted when there are no references to them. Note that the use of the global variable is not considered a good programming practice, but is shown here only to illustrate the concept of reference counting.

Additional global variable:

```
shared_ptr<Node> global_reference;
```

Modified code in `listTest`:

```
void listTest()
{
	shared_ptr<Node> root(new Node(10, nullptr));
	shared_ptr<Node> next1(new Node(20, nullptr));
	shared_ptr<Node> next2;
	// After a shared_ptr is declared we can set it
	// using the reset function
	next2.reset(new Node(30, nullptr));
	// Link the nodes together
	root->setNext(next1);
	next1->setNext(next2);

	// Output the list
	shared_ptr<Node> temp;
	temp = root;
	while (temp != nullptr)
	{
		cout << temp->getNum() << endl;
		temp = temp->getNext();
	}
	// The line below creates a reference to the second item
	// in the linked list
	global_reference = root->getNext();
}
```

Program output:

```
10
20
30
Deleting 10
Exiting program.
Deleting 20
Deleting 30
```

The big difference is that only the first node is deleted when the `listTest` function exits because it has no references. The remaining two nodes still have references due to the global variable. However, when the program finally exits, even these nodes go out of scope and memory is deallocated.

You should be aware that the `shared_ptr` class does not solve all of your problems. There is a problem if you make a circular list of references, in which case the reference count will never each 0 and memory will not be reclaimed. To solve this problem, C++11 includes an additional class named `weak_ptr` in which case an object will be destroyed if a `weak_ptr` is the only reference to it. As long as at least one of your links is conncted by a `weak_ptr` then the entire circular list will eventually be deallocated.

C++11 also includes a class named `unique_ptr` that cannot be assigned to any other pointer. Older versions of C++ supported a class named `auto_ptr` but it has been deprecated in C++11.