C++ Trees Part 1: Understanding the core::tree<> Implementation

by Justin Gottschlich

1. Introduction

For all of C++'s brilliance of strongly type-safe generic programming design, its standard library selection leaves much to be desired. Unlike Java, which has libraries for just about everything, C++ only has a handful. These C++ standard libraries are for containers, algorithms, streams and the like. Surprisingly, C++'s list of standard containers does not include a tree container [Musser1].

While hopes exist that C++0x may come with tree container support, C++'s current lack of native tree container support is enough to push programmers away from correctly designed systems using trees, in favor of poorly designed systems that use currently available standard containers. I created the core::tree<> container primarily to overcome this hurdle and add a missing piece to C++'s already powerful and elegantly designed generic containers.

After presenting the arguments made within this article to senior software engineers at Quark, Inc., they began adopting the core::tree<> family (tree, multitree, tree_pair, multitree_pair). Quark has since licensed the core::tree family (royalty free) and has been using it since 2002. They are currently using the core::tree family in their world class software, QuarkXPress 6.0 and expanding its use greatly for QuarkXPress 7.0 [Quark1].

This article is only part one of a series of articles written on the core::tree<> family. Part one primarily focuses on explaining the limitations of using C++'s map<> and multimap<> as tree containers and show the advantages of using the core::tree<> in its stead.

Additionally, simplistic sample core::tree<> code is given to increase basic familiarity with the core::tree<> design. Later installments of to the core::tree<> series will explain more complex usage of the core::tree<> and its flexibility in design.

Lastly, the core::tree<> source code is included for anyone to use, royalty free. The only request is that the licensing agreement found in the tree.h header is followed.

The layout of this article is as follows:

1. Introduction
2. std::map<> versus core::tree<>
3. Using the core::tree<>
4. Conclusion
5. References
6. core::tree<> source (tree.h header)

2. std::map<> versus core::tree<>

Many experienced C++ programmers use C++'s standard template library's (STL) map<> and multimap<> to generate trees. Yet, these containers make poor substitutes for true tree designs for a variety of reasons. This section will take a closer look at the pitfalls of using C++'s std::map<> and std::multimap<> for tree containment.

2.1 Basic std::map<> tree implementation

Before beginning the implementation of a tree using std::map<>, a framework must be defined in which the tree will work. For the purposes of this article the following namespace will be used to control the number of leaves generated at the root level and maximum branch depth of the trees constructed:

```
////////////////////////////////////////////////////////////////////////////////
namespace nTreeData
{
   const int kMaxLeaves = 10;
   const int kMaxDepth = 5;
}
```

Figure 2.1

Additionally, the following class defined (in figure 2.2) will be used as the leaf node, which will be inserted at every branch of the tree.

```
   ////////////////////////////////////////////////////////////////////////////////
   class Leaf
   {
   public:

      Leaf() : value_(0) {}
      explicit Leaf(const int &value) : value_(value) {}

      const int &value() const { return value_; }

      bool operator==(const Leaf &rhs) const { return this->value() == rhs.value(); }
      bool operator<(const Leaf &rhs) const { return this->value() < rhs.value(); }

   private:
      int value_;
   };
```

Figure 2.2

Combining the definitions provided in figure 2.1 and figure 2.2, an example of an std::map<> tree can be constructed:

```
   #include <map>
   #include <iostream>

   typedef std::map<Leaf, int> LeafMapConcrete;
   typedef std::map<Leaf, int>* LeafMapPointer;
   typedef std::map<Leaf, LeafMapPointer > LeafMap;

   void fun()
   {
      using namespace nTreeData;
      LeafMap leafTree;

      ////////////////////////////////////////////////////////////////////////////
      // create a simple leaf tree
      ////////////////////////////////////////////////////////////////////////////
      for (int i = 0; i < kMaxLeaves; ++i)
      {
         // insert a starter leaf
         LeafMapPointer p = new LeafMapConcrete;
         leafTree.insert(LeafMap::value_type(Leaf(i), p));
         LeafMap::iterator iter = leafTree.find(Leaf(i));

         // continue inserting children inside of children
         for (int depth = 0; depth < kMaxDepth; ++depth)
         {
            LeafMapPointer inner = new LeafMapConcrete;
            LeafMap* outer = (LeafMap*)(iter->second);
            outer->insert(LeafMap::value_type(Leaf(depth), inner));

            iter = outer->find(Leaf(depth));
         }
      }
```

```
    ////////////////////////////////////////////////////////////////////////////
    // deallocate the leaf tree
    ////////////////////////////////////////////////////////////////////////////
    for (LeafMap::iterator destroy = leafTree.begin(); destroy != leafTree.end();
         ++destroy)
    {
       LeafMap::const_iterator inner = destroy;
       LeafMap* iterMap = (LeafMap*)(destroy->second);
       LeafMap* lastMap;

       for (inner = iterMap->begin(); inner != iterMap->end(); inner =
            iterMap->begin())
       {
          lastMap = iterMap;
          // move the iterMap forward
          iterMap = (LeafMap*)inner->second;
          delete lastMap;
       }
    }
 }
```

Figure 2.3

Figure 2.3 demonstrates how to use a std::map<> to implement a tree in C++. The above implementation is a common method for overcoming STL's lack of native tree container support. Unfortunately, it has many problems:

1. Dynamic memory allocation and deallocation must be implemented for the tree by the programmer. Additional code must be added in figure 2.3 to do full depth and breadth tree iteration to allocate and deallocate each branch. Currently, figure 2.3 only provides simple deallocation (since we're assuming knowledge of the tree layout). Correct and complete memory deallocation requires more work. Additionally, it is very common for programmers to make mistakes when implementing complex memory management, which leads to memory leaks. One of the advantages of using STL containers is that they perform memory management internally [Josuttis1]. However, when using STL maps to construct trees in this manner, the memory management advantage of STL is lost.

2. Many C-style / reinterpret_cast<> type-casts are required. Type-casting is dangerous (especially, C-style casting and reinterpret_cast<>). Accidental incorrect type-casting can cause unexpected run time behavior. As Stroustrup reminds us, we should avoid explicit casting [Stroustrup1].

3. The code is complex. Simply doing the construction and destruction of the tree is rather hard to understand. If the code to generate and populate a simple tree is this difficult to write, how difficult will it be to write more complex trees? What about the difficulty of maintaining this tree code (especially by someone who didn't originally write the code)? Sutter and Alexandrescu point out in their "C++ Coding Standards", simple should always be preferred over complex [Sutter1]. Figure 2.3's tree implementation clearly violates this rule.

4. There is a great deal of wasted space. For each branch of the tree generated, an extra integer is generated serving no purpose except as a filler. Unfortunately, there is no way to remove this extra filler. This is another clue that the design is flawed - unnecessary pieces are present in the implementation that do not aid the design.

Consider now the same tree being generated using the core::tree<> container:

```
#include "tree.h"
#include <iostream>
```

```
void fun()
{
   using namespace nTreeData;
   using namespace core;

   core::tree<Leaf> leafTree;

   //////////////////////////////////////////////////////////////////////////
   // create a simple leaf tree
   //////////////////////////////////////////////////////////////////////////
   for (int i = 0; i < kMaxLeaves; ++i)
   {
      // insert a starter leaf
      tree<Leaf>::iterator iter = leafTree.insert(Leaf(i));

      // continue inserting children each time
      for (int depth = 0; depth < kMaxDepth; ++depth)
      {
         // insert and step into the newly created branch
         iter = iter.insert(Leaf(depth));
      }
   }
}
```

Figure 2.4

The code within figure 2.4 implements the same tree as the std::map<> solution within figure 2.3. However, the tree constructed using the core::tree<> container (figure 2.4) is less complex than the tree constructed using std::map<>. Furthermore, the core::tree<> implementation requires less code than the std::map<> implementation. Lastly, the core::tree<> implementation has none of the pitfalls the std::map<> implementation has:

1. No dynamic memory allocation or deallocation is needed, it is all handled internally.
2. Not even a single type-cast is used.
3. The code is very straight forward.
4. There is no wasted space.

Reviewing the above implementations of trees, it is clear the tree implementation using std::map<> is error-prone, complex and run time unsafe. However, the tree implementation using the core::tree<> is simple and elegant, and solves the tree design problem with no additional programmatic overhead. The core::tree<> code is also easier to understand than the std::map<> code which leads to easier code maintenance.

3. Using the core::tree<>

C++'s STL implementation is genius. The advantages and subtleties of its implementation are so numerous, it is impossible to cover them in a single article, or even in a single book. It was with STL in mind, that the core::tree<> was designed. The core::tree<> container follows as many of the STL design practices as possible, while still ensuring its own tree behavior remains correct.

The core::tree<> implements both const_iterators and iterators for tree iteration. By default, it uses operator<() and operator==() for inserts and finds/removes, but allows predicates to be used to overload that functionality. The core::tree<> implements begin() and end() on its containers as well as post-increment and pre-increment on its iterators. For moving in and out within the tree, methods in() and out() can be called – this makes tree depth iteration very simple. Many other powerful pieces of functionality are implemented as well, such as size(), level() and, of course, full tree copying (just like all STL containers) by use of operator=().

Perhaps the biggest downfall of the std::map<> tree implementation is its lack of simple "complete" tree copying. Calling operator=() on the std::map<> would result in pointer copying, not object copying and implementing a full tree copy would require even more dynamical memory allocation introducing more possibilities for erroneous programmer made memory management mistakes (more memory leaks).

Again, the above problems dissolve when using the core::tree<>. Copying a tree into another tree is as simple as:

```
core::tree<Leaf> tree1;
// ... do work on tree1 here
// now copy the entire contents of tree1 into tree2
core::tree<Leaf> tree2 = tree1;
```

Thus, the core::tree<> container's learning curve is very low for anyone who is already familiar with C++'s STL containers.

The below sample code demonstrates the ease of use of the core::tree<> container. The code in figure 3.1 performs simple tree construction and tree output.

```
#include <iostream>
#include "tree.h"

void fun()
{
   // the tree containers all sit inside the core namespace
   using namespace core;
   using namespace nTreeData;

   tree<Leaf> leafTree;

   ///////////////////////////////////////////////////////////////////////////
   // create a simple leaf tree
   ///////////////////////////////////////////////////////////////////////////
   for (int i = 0; i < kMaxLeaves; ++i)
   {
      // insert a starter leaf
      tree<Leaf>::iterator iter = leafTree.insert(Leaf(i));

      // continue inserting children each time
      for (int depth = 0; depth < kMaxDepth; ++depth)
      {
         // insert and step into the newly created branch
         iter = iter.insert(Leaf(depth));
      }
   }

   ///////////////////////////////////////////////////////////////////////////
   // output the leaf tree
   ///////////////////////////////////////////////////////////////////////////
   for (tree<Leaf>::const_iterator iter = leafTree.begin(); iter != leafTree.end();
   ++iter)
   {
      std::cout << iter.data().value() << std::endl;

      tree<Leaf>::const_iterator inner = iter;
      // tree's iterators are containers themselves - use the same iterator to
      // traverse inwards through the tree
      for (inner = inner.in(); inner != inner.end(); inner = inner.in())
      {
         for (int tabs = 1; tabs < inner.level(); ++tabs) std::cout << "\t";

         std::cout << (*inner).value() << std::endl;
      }
   }
}
```

Figure 3.1

For clarity, a step-by-step analysis of the core::tree<> code in figure 3.1 is given below.

1. Construct the tree container:

```
tree<Leaf> leafTree;
```

2. Populate the tree with data. The outer for loop inserts branches into the root tree. The inner for loop inserts branches into each additional branch inserted. Thus, the outer for loop is a breadth population and the inner for loop is a depth population. Notice that the inner for loop assigns the inserted iterator to itself: iter = iter.insert(). This forces iter to continue stepping inward, becoming the iterator of the branch inserted by the iter.insert() operation.

```
for (int i = 0; i < kMaxLeaves; ++i)
{
   // insert a starter leaf
   tree<Leaf>::iterator iter = leafTree.insert(Leaf(i));

   // continue inserting children each time
   for (int depth = 0; depth < kMaxDepth; ++depth)
   {
      // insert and step into the newly created branch
      iter = iter.insert(Leaf(depth));
   }
}
```

3. Iterate through the tree. The outer for loop iterates through the branches of the root tree. The inner for loop iterates inward. Again, notice how inner = inner.in() is performed within the nested for loop. This forces the inner iterator to continue to step into itself until there are no further branches inward.

```
for (tree<Leaf>::const_iterator iter = leafTree.begin(); iter != leafTree.end();
   ++iter)
{
   std::cout << iter.data().value() << std::endl;

   tree<Leaf>::const_iterator inner = iter;
   // tree's iterators are containers themselves - use the same iterator to
   // traverse inwards through the tree
   for (inner = inner.in(); inner != inner.end(); inner = inner.in())
   {
      for (int tabs = 1; tabs < inner.level(); ++tabs) std::cout << "\t";

      std::cout << (*inner).value() << std::endl;
   }
}
```

4. The output of the leafTree that's built above is as follows:

```
0
        0
                1
                        2
                                3
                                        4
1
        0
                1
                        2
                                3
                                        4
2
        0
                1
                        2
                                3
                                        4
3
        0
                1
                        2
                                3
                                        4
4
        0
                1
                        2
                                3
                                        4
5
        0
                1
                        2
                                3
                                        4
```

```
6
	0
		1
			2
				3
					4
7
	0
		1
			2
				3
					4
```

```
8
	0
		1
			2
				3
					4
9
	0
		1
			2
				3
					4
```

As you can see from the above output the root of the tree has ten branches. Each of those ten branches contain five inner branches, all nested within each other. Analyzing the output in step 4 can aid in understanding the code in step 3.

Due to the core::tree<> design following some of the STL container paradigms, certain STL algorithms can be followed. For example, the std::for_each algorithm can be used to iterate across any breadth of tree (keep in mind, tree depth iteration requires a bit more work). Figure 3.2 demonstrates a simple core::tree<> implementation using std::for_each.

```
#include <algorithm>
#include <iostream>
#include “tree.h”

//////////////////////////////////////////////////////////////////////////////
void outputLeaf(const Leaf &l)
{
   std::cout << l.value() << std::endl;
}

//////////////////////////////////////////////////////////////////////////////
void fun()
{
   core::tree<Leaf> leafTree;

   for (int i = 0; i < nTreeData::kMaxLeaves; ++i) leafTree.insert(Leaf(i));
   std::for_each(leafTree.begin(), leafTree.end(), outputLeaf);
}
```

Figure 3.2

To perform complete tree output iteration (both breadth and depth) in a single function, a minor amount of recursion is generally suggested (although it can be performed iteratively). However, the code is still very straightforward and easy to write:

```
#include <iostream>
#include “tree.h”

//////////////////////////////////////////////////////////////////////////////
void outputLeaf(core::tree<Leaf>::const_iterator &tree)
{
   // a tree iterator can check itself for its end
   for (core::tree<Leaf>::const_iterator i = tree.begin();i != tree.end(); ++i)
   {
      for (int tabs = 1; tabs < i.level(); ++tabs) std::cout << "\t";
      std::cout << (*i).value() << std::endl;

      outputLeaf(i);
   }
}

//////////////////////////////////////////////////////////////////////////////
void fun()
{
   // the tree containers all sit inside the core namespace
```

```
    using namespace core;
    using namespace nTreeData;

    core::tree<Leaf> leafTree;

    //for (int i = 0; i < nTreeData::kMaxLeaves; ++i) leafTree.insert(Leaf(i));
    ///////////////////////////////////////////////////////////////////////////
    // create a simple leaf tree
    ///////////////////////////////////////////////////////////////////////////
    for (int i = 0; i < kMaxLeaves; ++i)
    {
       // insert a starter leaf
       tree<Leaf>::iterator iter = leafTree.insert(Leaf(i));

       // continue inserting children each time
       for (int depth = 0; depth < kMaxDepth; ++depth)
       {
          // insert a 100 placeholder leaf, then insert a leaf and step into it
          iter.insert(Leaf(100));
          iter = iter.insert(Leaf(depth));
       }
    }

    outputLeaf(static_cast<core::tree<Leaf>::const_iterator >(leafTree));
 }
```

Figure 3.3

Figure 3.3 will properly output a tree with any number of branches within any given branch. Additionally, figure 3.3 demonstrates a fundamental difference between core::tree<> containers and STL containers:

core::trees<> ***are*** core::tree<>::iterators.

The last line of code in figure 3.3 shows the static_cast<> operation converting a core::tree<> into a core::tree<>const_iterator. This is a fundamental deviation and necessity for the core::tree<> implementation. This concept alone, is what makes the core::tree<> container possible. More detailed explanation is given about this concept in the next portion of the core::tree<> series.

4. Conclusion

C++'s STL containers are superior when implementing the role they were designed to perform. However, using STL containers to perform actions they weren't designed to perform (as with anything else) will not only result in a poorly designed systems, it will likely cause many implementation flaws to arise (as seen in section 2.1).

C++'s std::map<> was built for key/value tables, not for N-depth tree implementations. While programmers, especially C++ programmers, are notorious for inventive solutions, generic frameworks should be used in the manner they were meant to be used. Using these frameworks for purposes other than their intended roles may (and often will) result in trouble.

The core::tree<> container is far from perfect. It certainly has limitations and flaws in its implementation; it will not work with all STL algorithms and it may even have trouble compiling on some systems. Yet, when considering what is available in C++ today and the advantages the core::tree<> brings with it, it is a great replacement for std::map<> tree implementations. Additionally, any problems encountered can be fixed directly by those using it as the entire source is at your disposal.

While the core::tree<> implementation of trees can surely be improved (as with most anything) the core::tree<> is 1) easy for most C++ programmers familiar with STL to use and 2) is already being used in two commercial pieces of software: Quark XPress 6.0 and Nodeka. This shows, at the very least, its stability of operation. If you have the need for generic tree containers in your C++ software, you should consider using the core::tree<>.

5. References

[Josuttis1] Josuttis, Nicolai M. The C++ Standard Library. Addison-Wesley, Upper Saddle River, NJ 1999 . pp 31-32.

[Musser1] Musser, David and Atul Saini. STL Tutorial and Reference Guide. Addison-Wesley, Upper Saddle River, NJ 1996. pp 69.

[Quark1] Quark, Inc. currently uses the core::tree_pair and core::multitree_pair in their commercial software for tree implementations (mostly XML trees) as the core::tree and core::multitree weren't available at the time they licensed the software. Quark has recently (April 2004) requested increased licensing permission due to their expanding need of the core::tree family for its use anywhere within their commercial software.

[Stroustrup1] Stroustrup, Bjarne. The C++ Programming Language, Special Edition. Addison-Wesley, Upper Saddle River, NJ 1997. pp 417-425.

[Sutter1] Sutter, Herb and Andrei Alexandrescu. C++ Coding Standards. Addison-Wesley, Upper Saddle River, NJ 2005.

6. core::tree<> source

```
///////////////////////////////////////////////////////////////////////////////
//
// This generic tree container is the property of Justin Gottschlich. It may
// be used freely in commercial software or non-commercial software without
// explicit permission from Mr. Gottschlich. However this header file comment
// must always appear in this header file in its entirety.
//
// You may use this source code free of charge in any environment, pending
// you e-mail Justin (justin@nodeka.com) so he is aware of how the tree
// container is being used and send updates as they are made.
//
// (c) 1999-2004 Justin Gottschlich
//
///////////////////////////////////////////////////////////////////////////////

#ifndef tree_header_file
#define tree_header_file

#ifndef NULL
#define NULL 0
#endif

#if WIN32
#pragma warning( push )
// Disable warning for multiple operator= defines
#pragma warning( disable : 4522 )
#pragma warning( disable : 4786 )
#endif // WIN32

namespace core {

///////////////////////////////////////////////////////////////////////////////
// tree_iterator forward declaration
///////////////////////////////////////////////////////////////////////////////
template <typename T> class tree_iterator;

///////////////////////////////////////////////////////////////////////////////
// tree pair object definition
///////////////////////////////////////////////////////////////////////////////
template <typename T> class tree
{
public:

        typedef tree_iterator<T> iterator;
        typedef const tree_iterator<T> const_iterator;

private:

        // Class data
        mutable T data_;

        // What level are we on?
        mutable size_t level_;
        mutable size_t size_;

        // Nobody gets any access to this
        mutable tree *next_;
        mutable tree *prev_;
        mutable tree *out_;
        mutable tree *in_;

        ///////////////////////////////////////////////////////////////////////////
        // Removes a link to a node ... doesn't destroy the CTree, just rips it
        // out of it's current location. This is used so it can be placed elsewhere
        // without trouble.
        ///////////////////////////////////////////////////////////////////////////
        void disconnect_()
        {
```

```
		// unlink this from the master node
		if (this->out_ != NULL) {

			// this->out_ is going to be called alot in succession "register"
it
			register tree *out = this->out_;

			// Decrement the size of the outter level
			--(out->size_);

			if (out->in_ == this) {
				if (NULL == this->next_) {
					// If this is the last node of this level, zap the
hidden node
					delete this->prev_;
					out->in_ = NULL;
				}
				else {
					// Otherwise, just reattatch the head node to the
next node
					this->prev_->next_ = this->next_;
					this->next_->prev_ = this->prev_;
					out->in_ = this->next_;
				}
			}
			else {
				// We should be able to do this absolutely.
				this->prev_->next_ = this->next_;
				if (NULL != this->next_) this->next_->prev_ = this->prev_;
			}
		}
		// Point to nothing
		this->next_ = this->prev_ = NULL;
	}

	//////////////////////////////////////////////////////////////////////////
	// End of the tree list, private only
	//////////////////////////////////////////////////////////////////////////
	const tree* end_() const { return (NULL); }

	//////////////////////////////////////////////////////////////////////////
	// Does the actual insert into the tree
	//////////////////////////////////////////////////////////////////////////
	tree& i_insert(tree *inTree, tree *level, bool (*pObj)(tree*, tree*))
	{
		// Do NOT move this line beyond this point. The reason is because we must
		// check to see if the node exists here because we may be removing the
ONLY
		// node in the tree. If it is then NULL == level->in_. DO NOT REMOVE THIS
		//if (false == level->mDuplicates)

		// never allow duplicate keys
		level->remove(inTree->data());

		// if there's no inner tree, make it
		if (NULL == level->in_) {
			// Dummy node, create it -- if good memory do stuff, if NULL throw
			if (tree *temp = new tree) {
				temp->next_ = inTree;
				inTree->prev_ = temp;
				level->in_ = inTree;
			}
			else throw "allocation failed";
		}
		else {

			tree *temp = level->in_->prev_;

			while (true) {
				if (NULL == temp->next_) {
					temp->next_ = inTree;
```

```
                                        inTree->prev_ = temp;
                                        break;
                                }
                                else if ( pObj(inTree, temp->next_) ) {

                                        tree *hold = temp->next_;

                                        // temp -> inTree -> hold
                                        temp->next_ = inTree;
                                        inTree->next_ = hold;

                                        // temp <- inTree <- hold
                                        hold->prev_ = inTree;
                                        inTree->prev_ = temp;

                                        // If we just inserted on the first node, we need to
make sure
                                        // the in pointer goes to inTree
                                        if (hold == level->in_) level->in_ = inTree;
                                        break;
                                }
                                temp = temp->next_;
                        }
                }

                inTree->out_ = level;
                ++(level->size_);

                inTree->level_ = level->level() + 1;
                return (*inTree);
        }

        ////////////////////////////////////////////////////////////////////////////
        // No function object
        ////////////////////////////////////////////////////////////////////////////
        tree& i_insert(tree *inTree, tree *level)
        {
                // Do NOT move this line beyond this point. The reason is because we must
                // check to see if the node exists here because we may be removing the
ONLY
                // node in the tree. If it is then NULL == level->in_. DO NOT REMOVE THIS
                //if (false == level->mDuplicates)
                level->remove(inTree->data());

                // if there's no inner tree, make it
                if (NULL == level->in_) {
                        // Dummy node, create it -- if good memory do stuff, if NULL throw
                        if (tree *temp = new tree) {
                                temp->next_ = inTree;
                                inTree->prev_ = temp;
                                level->in_ = inTree;
                        }
                        else throw "allocation failed";
                }
                else {

                        tree *temp = level->in_->prev_;

                        while (true) {
                                if (NULL == temp->next_) {
                                        temp->next_ = inTree;
                                        inTree->prev_ = temp;
                                        break;
                                }
                                else if ( inTree->data() < temp->next_->data() ) {

                                        tree *hold = temp->next_;

                                        // temp -> inTree -> hold
                                        temp->next_ = inTree;
                                        inTree->next_ = hold;
```

```
                                        // temp <- inTree <- hold
                                        hold->prev_ = inTree;
                                        inTree->prev_ = temp;

                                        // If we just inserted on the first node, we need to
make sure
                                        // the in pointer goes to inTree
                                        if (hold == level->in_) level->in_ = inTree;
                                        break;
                                }
                                temp = temp->next_;
                        }
                }

                inTree->out_ = level;
                ++(level->size_);

                inTree->level_ = level->level() + 1;
                return (*inTree);
        }

protected:

        //////////////////////////////////////////////////////////////////////////
        //////////////////////////////////////////////////////////////////////////
        const size_t size(const tree& in) const { return in.size(); }
        const size_t level(const tree& in) const { return in.level(); }

        //////////////////////////////////////////////////////////////////////////
        // Points to the beginning of the list and sets the current
        //////////////////////////////////////////////////////////////////////////
        const_iterator begin(const tree& in) const { return iterator( *(in.in_) ); }

        //////////////////////////////////////////////////////////////////////////
        // Notice that we're returning a const tree* here and not an iterator.
        // This is because the iterator itself has a member to a pointer. Doing
        // an iterator constructor here would be less efficient than just
        // returning a tree* which can be assigned internally inside the iterator
        // operator--(). Also because no one can call prev from a tree itself
        // (this is protected), we don't have to worry about safety issues except
        // for iterator safety.
        //////////////////////////////////////////////////////////////////////////
        const tree* prev(const tree& in) const { return (in.prev_); }

        //////////////////////////////////////////////////////////////////////////
        // Notice that we're returning a const tree* here and not an iterator.
        // This is because the iterator itself has a member to a pointer. Doing
        // an iterator constructor here would be less efficient than just
        // returning a tree* which can be assigned internally inside the iterator
        // operator++(). Also because no one can call prev from a tree itself
        // (this is protected), we don't have to worry about safety issues except
        // for iterator safety.
        //////////////////////////////////////////////////////////////////////////
        const tree* next(const tree& in) const { return (in.next_); }

        //////////////////////////////////////////////////////////////////////////
        const_iterator in(const tree& in) const { return iterator( *(in.in_) ); }

        //////////////////////////////////////////////////////////////////////////
        const_iterator out(const tree& in) const { return iterator( *(in.out_) ); }

public:

        //////////////////////////////////////////////////////////////////////////
        // Default constructor
        //////////////////////////////////////////////////////////////////////////
        tree() : next_(0), prev_(0), in_(0), out_(0), level_(0), size_(0) {}

        //////////////////////////////////////////////////////////////////////////
        // Paired <T> constructor
```

```
	////////////////////////////////////////////////////////////////////////////
	tree(const T &inT) : data_(inT), next_(0), prev_(0), in_(0), out_(0), level_(0),
size_(0) {}

	////////////////////////////////////////////////////////////////////////////
	// operator==, expects operator== has been written for both t and u
	////////////////////////////////////////////////////////////////////////////
	const bool operator==(const tree &inTree) const
	{
		if (this->data_ == inTree.data_) return true;
		return false;
	}

	////////////////////////////////////////////////////////////////////////////
	// The operator= which is a real copy, hidden and undefined
	////////////////////////////////////////////////////////////////////////////
	const tree& operator=(const tree& in)
	{
		this->clear();

		this->data_ = in.data_;
		this->copy_tree(in);

		return *this;
	}

	////////////////////////////////////////////////////////////////////////////
	// copy constructor - now visible
	////////////////////////////////////////////////////////////////////////////
	tree(const tree &in) : data_(in.data_), next_(0), prev_(0), in_(0), out_(0),
		level_(0), size_(0) { *this = in; }

	////////////////////////////////////////////////////////////////////////////
	// destructor -- cleans out all branches, destroyed entire tree
	////////////////////////////////////////////////////////////////////////////
	virtual ~tree()
	{
		// Disconnect ourselves -- very important for decrementing the
		// size of our parent
		this->disconnect_();

		// Now get rid of our children -- but be smart about it,
		// right before we destroy it set it's out_ to NULL
		// that way Disconnect fails immediately -- much faster

		if (this->size() > 0) {
			register tree *cur = this->in_, *prev = this->in_->prev_;

			// Delete the head node
			prev->out_ = NULL;
			delete prev;

			for (; this->size_ > 0; --this->size_) {

				prev = cur;
				cur = cur->next_;

				prev->out_ = NULL;
				delete prev;
			}
		}
	}

	////////////////////////////////////////////////////////////////////////////
	////////////////////////////////////////////////////////////////////////////
	void copy_tree(const tree& in)
	{
		// for each branch iterate through all nodes and copy them
		for (iterator i = in.begin(); in.end() != i; ++i) {
			iterator inserted = this->insert(i.data());
```

```
                // for each node, see if there are inners - if so, copy those too
                if (i.size() != 0) inserted.tree_ptr()->copy_tree(*i.tree_ptr());
        }
}

//////////////////////////////////////////////////////////////////////////
// Returns the first element of our tree
//////////////////////////////////////////////////////////////////////////
const_iterator begin() const { return iterator( *(this->in_) ); }
iterator begin() { return iterator( *(this->in_) ); }

//////////////////////////////////////////////////////////////////////////
// Returns end_of_iterator
//////////////////////////////////////////////////////////////////////////
const_iterator& end() const { return tree::iterator::end_iterator(); }

//////////////////////////////////////////////////////////////////////////
// Returns the first element of our tree
//////////////////////////////////////////////////////////////////////////
const_iterator in() const { return iterator( *(this->in_) ); }
iterator in() { return iterator( *(this->in_) ); }

//////////////////////////////////////////////////////////////////////////
// Returns an iterator which steps out one level
//////////////////////////////////////////////////////////////////////////
const_iterator out() const { return iterator( *(this->out_) ); }
iterator out() { return iterator( *(this->out_) ); }

//////////////////////////////////////////////////////////////////////////
// much like destructor with the exception that empty can be called from
// an iterator, calling delete on an iterator will only delete the iterator
// calling empty from an iterator will delete the tree it's iterating.
//////////////////////////////////////////////////////////////////////////
void clear()
{
        // Now get rid of our children -- but be smart about it,
        // right before we destroy it set it's out_ to NULL
        // that way disconnect_ fails immediately, much faster
        if (this->size() > 0) {
                register tree *cur = this->in_, *prev = this->in_->prev_;

                // Delete the head node
                prev->out_ = NULL;
                delete prev;

                for (; this->size_ > 0; --this->size_) {

                        prev = cur;
                        cur = cur->next_;

                        prev->out_ = NULL;
                        delete prev;
                }

                // Set our inner pointer and our size to 0
                this->in_ = NULL;
                this->size_ = 0;
        }
}

//////////////////////////////////////////////////////////////////////////
// retrieves a const ref for the t member of the pair
//////////////////////////////////////////////////////////////////////////
T& data() { return this->data_; }

//////////////////////////////////////////////////////////////////////////
// sets and retrieves the t member of the pair
//////////////////////////////////////////////////////////////////////////
const T& data(const T &data_) const { return (this->data_ = data_); }

//////////////////////////////////////////////////////////////////////////
```

```
//
//////////////////////////////////////////////////////////////////////////
const size_t level() const { return (this->level_); }

//////////////////////////////////////////////////////////////////////////
//
//////////////////////////////////////////////////////////////////////////
const size_t size() const { return this->size_; }

//////////////////////////////////////////////////////////////////////////
// This creates a new tree node from parameters and then inserts it
// Also takes a function object which can be used for sorting on inserts
//////////////////////////////////////////////////////////////////////////
const iterator insert(const T &inT, bool (*pObj)(tree*, tree*))
{
	tree *myPair = new tree(inT);
	if (NULL == myPair) throw "allocation failed";
	return iterator(i_insert(myPair, this, pObj));
}

//////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
const iterator insert(const iterator &i)
{
	tree *myPair = new tree(i.data());
	if (NULL == myPair) throw "allocation failed";

	return iterator(i_insert(myPair, this));
}

//////////////////////////////////////////////////////////////////////////
// Insert with no function object
//////////////////////////////////////////////////////////////////////////
const iterator insert(const T &inT)
{
	tree *myPair = new tree(inT);
	if (NULL == myPair) throw "allocation failed";
	return iterator(i_insert(myPair, this));
}

//////////////////////////////////////////////////////////////////////////
// This takes an existing node, disconnects it from wherever it is, and then
// inserts it into a different tree. This does not create a new node from the
// passed in data.
//////////////////////////////////////////////////////////////////////////
const iterator reinsert(tree *in, bool (*pObj)(tree*, tree*))
{
	in->disconnect_();
	return iterator(i_insert(in, this, pObj));
}

//////////////////////////////////////////////////////////////////////////
// Reinsert with no function object
//////////////////////////////////////////////////////////////////////////
const iterator reinsert(tree *in)
{
	in->disconnect_();
	return iterator(i_insert(in, this));
}

//////////////////////////////////////////////////////////////////////////
// removes first matching t, returns true if found, otherwise false
//////////////////////////////////////////////////////////////////////////
const bool remove(const T &inData)
{
	if (tree *temp = this->in_) {
		do {
			if (inData == temp->data_) {
				delete temp;
				return true;
			}
```

```
                } while (NULL != (temp = temp->next_) );
        }
        return false;
}

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
const bool erase(const iterator& i)
{
        if (tree *temp = this->in_) {
                do {
                        if (temp == i.tree_ptr()) {
                                delete temp;
                                return true;
                        }
                } while (NULL != (temp = temp->next_) );
        }
        return false;
}

////////////////////////////////////////////////////////////////////////////
const iterator operator[](size_t loc) const
{
tree *temp;
        for (temp = this->in_; loc > 0; --loc) temp = temp->next_;
        return iterator(*temp);
}

////////////////////////////////////////////////////////////////////////////
iterator operator[](size_t loc)
{
tree *temp;
        for (temp = this->in_; loc > 0; --loc) temp = temp->next_;
        return iterator(*temp);
}

////////////////////////////////////////////////////////////////////////////
// internal_only interface, can't be called even with derived objects due
// to its direct reference to tree's private members
////////////////////////////////////////////////////////////////////////////
const iterator find(const T &inT) const
{
        if (tree *temp = this->in_) {
                do {
                        if (inT == temp->data_) return ( iterator(*temp) );
                } while (NULL != (temp = temp->next_) );
        }
        return tree::iterator::end_iterator();
}

////////////////////////////////////////////////////////////////////////////
//
////////////////////////////////////////////////////////////////////////////
const iterator find(const T &inT, bool (*obj)(const T&, const T&)) const
{
        if (tree *temp = this->in_) {
                do {
                        if ( obj(inT, temp->data_) ) return ( iterator(*temp) );
                } while (NULL != (temp = temp->next_) );
        }
        return tree::iterator::end_iterator();
}

////////////////////////////////////////////////////////////////////////////
// internal_only interface, can't be called even with derived objects due
// to its direct reference to tree's private members
////////////////////////////////////////////////////////////////////////////
const iterator find(const T &inT, const iterator &iter) const
{
        if (tree *temp = iter.tree_ptr()->next_) {
                do {
```

```
				if (inT == temp->data_) return ( iterator(*temp) );
			} while (NULL != (temp = temp->next_) );
		}
		return tree::iterator::end_iterator();
	}

	//////////////////////////////////////////////////////////////////////////
	//
	//////////////////////////////////////////////////////////////////////////
	const iterator find(const T &inT, const iterator &iter, bool (*obj)(const T&,
const T&)) const
	{
		if (tree *temp = iter.tree_ptr()->next_) {
			do {
				if ( obj(inT, temp->data_) ) return ( iterator(*temp) );
			} while (NULL != (temp = temp->next_) );
		}
		return tree::iterator::end_iterator();
	}
};

//////////////////////////////////////////////////////////////////////////
// Iterator for the tree
//
// Derived from tree<> only so iterator can access tree's protected
// methods directly and implement them in the way they make sense for the
// iterator
//
// The actual tree base members are never used (nor could they be since they
// are private to even iterator). When a tree object is created an "iterator"
// object is automatically created of the specific type. Thus forming the
// perfect relationship between the tree and the iterator, also keeping the
// template types defined on the fly for the iterator based specifically on
// the tree types which are being created.
//////////////////////////////////////////////////////////////////////////
template <typename T>
class tree_iterator : private tree<T>
{
private:
	typedef tree<T> TreeType;

	mutable TreeType *current_;

	static tree_iterator end_of_iterator;

	//////////////////////////////////////////////////////////////////////////
	// unaccessible from the outside world
	//////////////////////////////////////////////////////////////////////////
	TreeType* operator&();
	const TreeType* operator&() const;

public:

	TreeType* tree_ptr() const { return current_; }

	//////////////////////////////////////////////////////////////////////////
	// Returns the end_of_iterator for this <T,U,V> layout, this really speeds
	// up things like if (iter != tree.end() ), for (;iter != tree.end(); )
	//////////////////////////////////////////////////////////////////////////
	static const iterator& end_iterator() { return end_of_iterator; }

	//////////////////////////////////////////////////////////////////////////
	// Default constructor
	//////////////////////////////////////////////////////////////////////////
	tree_iterator() : current_(NULL) {}

	//////////////////////////////////////////////////////////////////////////
	// Copy constructors for iterators
	//////////////////////////////////////////////////////////////////////////
	tree_iterator(const tree_iterator& i) : current_(i.current_) {}
```

```
/////////////////////////////////////////////////////////////////////////////
// Copy constructor for trees
/////////////////////////////////////////////////////////////////////////////
tree_iterator(TreeType &tree_ref) : current_(&tree_ref) {}

/////////////////////////////////////////////////////////////////////////////
// Operator= for iterators
/////////////////////////////////////////////////////////////////////////////
const iterator& operator=(const tree_iterator& iter)
{
        this->current_ = iter.current_;
        return (*this);
}

/////////////////////////////////////////////////////////////////////////////
// Operator= for iterators
/////////////////////////////////////////////////////////////////////////////
const iterator& operator=(const tree_iterator& iter) const
{
        this->current_ = iter.current_;
        return (*this);
}

/////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////
const iterator operator[](size_t loc) const
{ return *(this->current_)[loc]; }

/////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////
iterator operator[](size_t loc)
{ return *(this->current_)[loc]; }

/////////////////////////////////////////////////////////////////////////////
// Operator= for trees
/////////////////////////////////////////////////////////////////////////////
const tree_iterator& operator=(const TreeType& rhs)
{
        this->current_ = &(const_cast< TreeType& >(rhs) );
        return (*this);
}

/////////////////////////////////////////////////////////////////////////////
// Destructor
/////////////////////////////////////////////////////////////////////////////
~tree_iterator() {};

/////////////////////////////////////////////////////////////////////////////
// Operator equals
/////////////////////////////////////////////////////////////////////////////
const bool operator==(const tree_iterator& rhs) const
{
        if (this->current_ == rhs.current_) return true;
        return false;
}

/////////////////////////////////////////////////////////////////////////////
// Operator not equals
/////////////////////////////////////////////////////////////////////////////
const bool operator!=(const tree_iterator& rhs) const
{ return !(*this == rhs); }

/////////////////////////////////////////////////////////////////////////////
// operator++, prefix
/////////////////////////////////////////////////////////////////////////////
const iterator& operator++() const
{
        this->current_ = ( const_cast< TreeType* >
                ( this->TreeType::next( *current_ ) ) );
        return (*this);
}
```

```
	/////////////////////////////////////////////////////////////////////////
	// operator++, postfix
	/////////////////////////////////////////////////////////////////////////
	const iterator operator++(int) const
	{
		iterator iTemp = *this;
		++(*this);
		return (iTemp);
	}

	/////////////////////////////////////////////////////////////////////////
	// operator--
	/////////////////////////////////////////////////////////////////////////
	iterator& operator--()
	{
		this->current_ = ( const_cast< TreeType* >
			( this->TreeType::prev( *current_ ) ) );
		return (*this);
	}

	/////////////////////////////////////////////////////////////////////////
	// Begin iteration through the tree
	/////////////////////////////////////////////////////////////////////////
	const iterator begin() const { return this->TreeType::begin( *current_ ); }
	iterator begin() { return this->TreeType::begin( *current_ ); }

	/////////////////////////////////////////////////////////////////////////
	// Return the in iterator of this tree
	/////////////////////////////////////////////////////////////////////////
	const iterator in() const { return this->TreeType::in( *current_ ); }
	iterator in() { return this->TreeType::in( *current_ ); }

	/////////////////////////////////////////////////////////////////////////
	// Return the out iterator of this tree
	/////////////////////////////////////////////////////////////////////////
	const iterator out() const { return this->TreeType::out( *current_ ); }
	iterator out() { return this->TreeType::out( *current_ ); }

	/////////////////////////////////////////////////////////////////////////
	// Are we at the end?
	/////////////////////////////////////////////////////////////////////////
	const iterator& end() const { return this->TreeType::end(); }

	/////////////////////////////////////////////////////////////////////////
	// Return the next guy
	/////////////////////////////////////////////////////////////////////////
	const iterator next() const
	{ return iterator (* const_cast< TreeType* >( this->TreeType::next( *current_ ) )
); }

	/////////////////////////////////////////////////////////////////////////
	// Insert into the iterator's mTree
	/////////////////////////////////////////////////////////////////////////
	const iterator insert(const T& t) const
	{ return this->current_->TreeType::insert(t); }

	/////////////////////////////////////////////////////////////////////////
	// Insert into the iterator's mTree (with a function object)
	/////////////////////////////////////////////////////////////////////////
	const iterator insert(const T& t, bool (*obj)(TreeType*, TreeType*)) const
	{ return this->current_->TreeType::insert(t, obj); }

	/////////////////////////////////////////////////////////////////////////
	// This takes an existing node, disconnects it from wherever it is, and then
	// inserts it into a different tree. This does not create a new node from the
	// passed in data.
	/////////////////////////////////////////////////////////////////////////
	const iterator reinsert(const iterator &in, bool (*obj)(TreeType*, TreeType*))
const
	{ return this->current_->TreeType::reinsert(in.current_, obj); }
```

```
        //////////////////////////////////////////////////////////////////////////
        // Reinsert with no function object
        //////////////////////////////////////////////////////////////////////////
        const iterator reinsert(const iterator &in) const
        { return this->current_->TreeType::reinsert(in.current_); }

        //////////////////////////////////////////////////////////////////////////
        // get the data of the iter
        //////////////////////////////////////////////////////////////////////////
        T& operator*() { return this->current_->data(); }
        const T& operator*() const { return this->current_->data(); }

        //////////////////////////////////////////////////////////////////////////
        // gets the t and u members of the current tree
        //////////////////////////////////////////////////////////////////////////
        T& data() { return this->current_->data(); }
        const T& data() const { return this->current_->data(); }

        //////////////////////////////////////////////////////////////////////////
        // sets and retrieves the t and u members of the pair
        //////////////////////////////////////////////////////////////////////////
        const T& data(const T &inData) const { return this->current_->data(inData); }

        //////////////////////////////////////////////////////////////////////////
        // Get the size of the current tree_iter
        //////////////////////////////////////////////////////////////////////////
        const size_t size() const { return this->TreeType::size( *current_ ); }

        //////////////////////////////////////////////////////////////////////////
        //
        //////////////////////////////////////////////////////////////////////////
        const size_t level() const { return this->TreeType::level( *current_ ); }

        //////////////////////////////////////////////////////////////////////////
        // Removes the first instance of T in the tree
        //////////////////////////////////////////////////////////////////////////
        const bool remove(const T &inT) const { return current_->remove(inT); }

        //////////////////////////////////////////////////////////////////////////
        // Finds the first instance of T in the tree
        //////////////////////////////////////////////////////////////////////////
        const iterator find(const T &inT) const { return current_->find(inT); }

        const iterator find(const T &inT, bool (*obj)(const T&, const T&)) const
        { return current_->find(inT, obj); }

        //////////////////////////////////////////////////////////////////////////
        // Finds the next instance of T based on the iterator passed in
        //////////////////////////////////////////////////////////////////////////
        const iterator find(const T &inT, const iterator &iter) const
        { return current_->find(inT, iter); }

        const iterator find(const T &inT, const iterator &iter,
               bool (*obj)(const T&, const T&)) const
        { return current_->find( inT, iter, obj ); }

        //////////////////////////////////////////////////////////////////////////
        // Empty this entire tree
        //////////////////////////////////////////////////////////////////////////
        void clear_tree() const { delete this->current_; this->current_ = NULL; }

        //////////////////////////////////////////////////////////////////////////
        // Empty this tree's children
        //////////////////////////////////////////////////////////////////////////
        void clear_children() const { this->current_->clear(); }
};

//////////////////////////////////////////////////////////////////////////
// Static iterator initialization
//////////////////////////////////////////////////////////////////////////
```

```
template <typename T>
tree_iterator<T> tree_iterator<T>::end_of_iterator;

};

#if WIN32
#pragma warning( pop )
#endif // WIN32

#endif // tree_header_file
```