Frog Data Structures

Rather than using STL data structures like std::vector, std::list, and std::map, Frog has its own. This is important on some platforms, because STL doesn't give us enough control over how, when, and where memory is allocated and deallocated from the underlying heap or heaps. When you're making a game for a platform with barely enough RAM, the heap can get fragmented if you aren't careful with allocation and deallocation. This can lead to noticeable slowdown and crashing. Unlike std::list and std::map, Frog's List and Map have Reserve methods to help avoid fragmentation. Frog also has its own implementations of heaps to help deal with these problems and with other quirks of quirky platforms. Luckily, if you're only concerned with Windows and Mac, you probably have so much RAM that you won't need to do anything tricky with special heaps or special allocators to avoid fragmentation. However, special heaps can still be useful on a PC for debugging memory corruption problems and finding memory leaks.

There are 5 main data structures in Frog: Table, TableStatic, List, ListStatic, and Map. Table is a managed array. TableStatic is a lot like Table, except that it doesn't make allocations from the heap. Instead, the space used for the array is part of the TableStatic instance itself. List is a doubly-linked list. Much like TableStatic, ListStatic is a special version of List that avoids heap allocations. Map is a self-balancing binary tree. Both Tables and Lists have additional helper functions for using them as queues or stacks: Push, Pop, Peek, Enqueue, and Dequeue.

The most important data structure to know is Table. It's the easiest, and therefore the least likely to waste your time or introduce bugs. For these reasons, it's usually still better to use a Table than a List or Map when List or Map would only run maybe negligibly faster. It's also often better to use a Table than a regular C array, because Table keeps track of the number of items, it can expand if needed, and it has a lot of helper functions.

There are still cases where the other data structures are preferable though. List and Map can be nice for avoiding memory fragmentation and for when you need the performance characteristics of doubly-linked lists and self-balancing binary trees. TableStatic and ListStatic are normally only used for special, low-level situations where the heap isn't available or isn't practical.

All of the data structures use templates to specify the type of value it's supposed to contain. Map has separate template parameters for the key and value types. Map also has a type definition for the comparator used to sort the items based on the keys, but this can typically be left at its default. TableStatic and ListStatic have a template parameter for the maximum capacity. Table also has a template parameter for "ExpansionPolicy" which should be left at its default. It's for controlling how much extra memory to allocate when allocating space for a new item.

To use a Frog data structure (and many other objects from Frog), you need to explicitly call Init before doing anything else with it. When you're done with the data structure, you need to explicitly call Deinit or it will leak memory. You can't rely entirely on the constructor and destructor. Separating initialization/deinitalization from allocation/deallocation gives us more control over what happens when. For example, there are a lot of singletons in Frog, and we want to be able to explicitly control the order in which they're initialized and deinitialized. This separation also allows us to reuse objects, including those that are statically allocated.

You should generally avoid passing Frog data structures by value. Instead, use pointers or C++ references. It's also probably safer if the types of objects in the container are either pointers or primitives (ints, bools, floats, etc.). For example, use Table<BigComplicatedObject*> rather than Table<BigComplicatedObject>.

By default, Frog's data structures are not easy to view in the Visual Studio debugger. This can be fixed by following the steps in the Frog Programmer Setup article on the wiki.

Table

Here are some simple examples of how to use Table.

```
// Initialize the collection.
Table<int> tableOfInts;
tableOfInts.Init();
```

```
   // Add some items.
   tableOfInts.AddBack(10);
   tableOfInts.AddBack(9);
   tableOfInts.AddBack(6);
   tableOfInts.AddBack(5);
   tableOfInts.AddBack(2);
   tableOfInts.AddBack(1);
   tableOfInts.AddBack(8);
   tableOfInts.AddBack(7);
   tableOfInts.AddBack(4);
   tableOfInts.AddBack(3);

   // Remove the odd numbers.  Iterate backwards so we don't need to think
   // about whether to advance the index number, depending on whether something
   // was removed.
   for(int index = tableOfInts.SizeGet() - 1; index >= 0; index--)
   {
      if(tableOfInts[index] % 2)
         tableOfInts.RemoveIndex(index);
   }

   // Sort them into ascending order using the < operator.
   tableOfInts.Sort();

   // Find the index of the number 4.  This will return -1 if it's missing.
   int indexOf4 = tableOfInts.FindIndex(4);
   if(indexOf4 != -1)
      DebugPrintf("index of 4: %d\n", indexOf4);
   else
      DebugPrintf("4 not found\n");

   // Print the sorted list.
   for(int index = 0; index < tableOfInts.SizeGet(); index++)
      DebugPrintf("%d\n", tableOfInts[index]);

   // Clean up.
   tableOfInts.Deinit();
```

By default, the Sort method uses the < operator to compare the items in the list. This is fine for getting numbers into ascending order, but if you want to compare objects, you probably aren't interested in sorting them by their address in memory. There could also be multiple ways you need to sort a set of items. For example, if your items are playing cards, you may want to sort them first in ascending order of rank, then by suit. Maybe you want ace to be the lowest card sometimes and other times the highest. For each of these situations, you can make a comparator function and pass it to Sort. The function should take two items as parameters and return true if first item should definitely go before the second item. It should return false if the second item should go first or if the two items are equivalent for sorting purposes. For example, here is a function that sorts cards first in descending order of rank, with aces low, then by suit.

```
bool ComparatorDescendingRankSuitAceLow(Card* cardA, Card* cardB)
{
   if(cardA->rank > cardB->rank)
```

```
      return true;
   if(cardA->rank < cardB->rank)
      return false;

   return cardA->suit < cardB->suit;
}
```

If the comparator function is a member of a class, it needs to be a static member. It's also possible to use a function object (functor) as a comparator, but a plain function is typically more convenient.

If you want to sort C strings alphabetically, again you can't rely on the < operator. Instead, you can wrap strcmp to make a simple comparator function. Better yet, you can use the one included in Frog. It's called StringComparator.

Map

Using a Map is more complicated, and it isn't appropriate for as many situations. However, it's faster for large indexed collections. If your keys are strings, it adds yet more work since you'll probably need to allocate and deallocate those strings when an item is added or deleted. Be mindful of how you handle keys when replacing a value without changing the key. Because of these complications, Maps are often used indirectly. For example, JSONValue uses a Map internally for the items in a JSON object, but someone using JSONValue doesn't need to worry about managing keys.

Here are some simple examples of how to use Map. (If you're using an old version of Frog, you might not be able to add the values here as literals. You'd need to use an intermediate variable instead.)

```
   // Set up a typedef so we don't need to keep typing the whole thing later
   // when using iterators.
   typedef Map<const char*, int> StringIntMap;

   // Initialize the collection.  Items will be sorted alphabetically.
   StringIntMap mapOfStringsToInts;
   mapOfStringsToInts.Init(StringComparator);

   // Add some items.  (Use the wrong value for Three on purpose.)
   mapOfStringsToInts.Add(StringClone("One"), 1);
   mapOfStringsToInts.Add(StringClone("Three"), 5);
   mapOfStringsToInts.Add(StringClone("Four"), 4);
   mapOfStringsToInts.Add(StringClone("Five"), 5);
   mapOfStringsToInts.Add(StringClone("Seven"), 7);
   mapOfStringsToInts.Add(StringClone("Nine"), 9);

   // Find and fix the value for "Three".  If nothing is found for that key,
   // the WithinCheck method of the iterator will return false.
   StringIntMap::Iterator threeIterator = mapOfStringsToInts.Find("Three");
   if(threeIterator.WithinCheck())
      threeIterator.Value() = 3;

   // Remove "Four", and remember to clean up its key after removal.
   StringIntMap::Iterator fourIterator = mapOfStringsToInts.Find("Four");
   if(fourIterator.WithinCheck())
   {
      const char* key = fourIterator.Key();
      mapOfStringsToInts.Remove(fourIterator);
      StringDelete(key);
   }

   // Iterate over the collection and print the items.
   for(StringIntMap::Iterator iterator = mapOfStringsToInts.Begin();
iterator.WithinCheck(); iterator.Next())
   {
      const char* key = iterator.Key();
      if(key)
         DebugPrintf("%s: %d\n", key, iterator.Value());
   }

   // Clean up.
   for(StringIntMap::Iterator iterator = mapOfStringsToInts.Begin();
iterator.WithinCheck();)
   {
      // Be sure to delete the keys after removal.
      const char* key = iterator.Key();
      mapOfStringsToInts.Remove(iterator);
      StringDelete(key);
   }
   mapOfStringsToInts.Deinit();
```

List

The interface for Lists is like a cross between Table and Map. Like Table, you don't necessarily need to worry about keys or comparators. Like Map, iterators are used to traverse the collection and make changes.

```
   // Initialize the collection.
   List<int> listOfInts;
   listOfInts.Init();

   // Add some items.
   listOfInts.Add(1);
   listOfInts.Add(2);
   listOfInts.Add(3);
   listOfInts.Add(4);
   listOfInts.Add(5);

   // Remove the even numbers.
   for(List<int>::Iterator iterator = listOfInts.Begin(); iterator.WithinCheck();)
   {
      if(iterator.Value() % 2)
         iterator.Next();
      else
         listOfInts.Remove(iterator);
   }

   // Print the remaining numbers.
   for(List<int>::Iterator iterator = listOfInts.Begin(); iterator.WithinCheck();
iterator.Next())
      DebugPrintf("%d\n", iterator.Value());
```

```
    // Clean up.
    listOfInts.Deinit();
```

Heaps and Allocators

Frog projects often have multiple heaps set up, and these heaps have corresponding allocator objects. These allocator objects can be passed to the Init method of a data structure control where the collection is getting its memory. The default heap is often fine to use. However, you sometimes need to make a bunch of alternating short-term allocations and long-term allocations. If both the short-term and long-term allocations are made from the same heap, you'll have fragmentation after freeing the short-term ones. To avoid this, you could make your short-term allocations from the designated temporary heap. For example, initializing a pair of Tables for this would look like the following.

```
    Table<int> longTermTable;
    Table<int> shortTermTable;
    longTermTable.Init();
    shortTermTable.Init(theAllocatorTemp);
```