```
// circularArrayList-inl.h:  An inline implementation file for the templated
// CircularArrayList class.

#include <stdexcept>

/*Constructor - initializes the size and starting headPos to 0,
 * allocates space for an array of size 10
 */
template <typename T>
CircularArrayList<T>::CircularArrayList() {
  headPos = 0;
  size = 0;
  capacity = 10;             // We've chosen an initial capacity of 10.
  values = new T[capacity];  // Allocates an array of 10 items on the heap.
}

/*Destructor - deallocates array from heap
 */
template <typename T>
CircularArrayList<T>::~CircularArrayList() {
  delete [] values;
}

/*getSize - returns number of elements stored in the list
 */
template <typename T>
int CircularArrayList<T>::getSize() {
  return size;
}

/*isEmpty - returns true iff (if and only if) there are no items in the list
 */
template <typename T>
bool CircularArrayList<T>::isEmpty() {
  return size == 0;
}

/*peekHead - returns item stored at the beginning of the list
 */
template <typename T>
T CircularArrayList<T>::peekHead() {
  if (isEmpty()) {
    throw std::runtime_error("Attempted to peekHead on an empty list.");
  }
  return values[headPos];
}

/*peekTail - returns last item stored in the list
 */
template <typename T>
T CircularArrayList<T>::peekTail() {
  if (isEmpty()) {
    throw std::runtime_error("Attempted to peekTail on an empty list.");
  }
  return values[(headPos+size-1)%capacity];
}

/*get - returns the item at position i in the list
 * @param i - an integer representing the position of the item relative to the
```

```
 *            beginning of the list
 */
template <typename T>
T CircularArrayList<T>::get(int i) {
  if (i < 0 || i >= size) {
    throw std::runtime_error("Attempted to get out of bounds.");
  }
  return values[(headPos+i)%capacity];
}

/*insertAtHead - adds an item to the beginning of the list
 * @param value - the item to add the list
 */
template <typename T>
void CircularArrayList<T>::insertAtHead(T value) {
  if (size == capacity) {
    expandCapacity();
  }
  headPos = (headPos+capacity-1) % capacity;  // Avoids mod of negative #.
  values[headPos] = value;    // Copies the value to the new first position.
  ++size;
}

/*insertAtTail - adds an item to the end of the list
 * @param value - the item to add the list
 */
template <typename T>
void CircularArrayList<T>::insertAtTail(T value) {
  if (size == capacity) {
    expandCapacity();
  }
  values[(headPos+size)%capacity] = value;
  ++size;
}

/*removeHead - removes and returns the first item in the list
 * @return the value of the item removed from the list
 */
template <typename T>
T CircularArrayList<T>::removeHead() {
  if (isEmpty()) {
    throw std::runtime_error("Attempted to removeHead on an empty list.");
  }

  int oldHeadPos = headPos;        // Store position of value to return
  headPos = (headPos+1)%capacity;  // Resets head to new position
  --size;
  return values[oldHeadPos];
}

/*removeTail - removes and returns the last item in the list
 * @return: the value of the item removed from the list
 */
template <typename T>
T CircularArrayList<T>::removeTail() {
  if (isEmpty()) {
    throw std::runtime_error("Attempted to removeTail on an empty list.");
  }
```

```
  --size;
  return values[(headPos+size)%capacity];
}

/*expandCapacity - private method for doubling the capacity of values
 */
template <typename T>
void CircularArrayList<T>::expandCapacity() {
   int newCapacity = 2*capacity;
   T* newArray = new T[newCapacity];
   for (int i = 0; i < capacity; ++i) {
     newArray[i] = values[(headPos + i) % capacity];
   }
   delete [] values;
   values = newArray;
   headPos = 0;
   capacity = newCapacity;
}
```