Vector.h

```
/***********************************************************************
* Header:
*    Vector
* Summary:
*    This class contains the notion of a container: a bucket to hold
*    data for the user. This is just a starting-point for more advanced
*    constainers such as the vector, set, stack, queue, deque, and map
*    which we will build later this semester.
*
*    This will contain the class definition of:
*        Vector         : A class that holds stuff
*        VectorIterator : An interator through Vector
*        VectorConstIterator : A const iterator
* Author
*    Br. Helfrich
************************************************************************/

#ifndef CONTAINER_H
#define CONTAINER_H

#include <cassert>
#include <iostream>

// forward declaration for VectorIterator
template <class T>
class VectorIterator;

// forward declaration for VectorConstIterator
template <class T>
class VectorConstIterator;

/*****************************************
 * CONTAINER
 * A class that holds stuff
 *****************************************/
template <class T>
class Vector
{
public:
   // default constructor : empty and kinda useless
   Vector() : numItems(0), vCapacity(0), data(NULL) {}

   // copy constructor : copy it
   Vector(const Vector & rhs) throw (const char *);

   // non-default constructor : pre-allocate
   Vector(int vCapacity) throw (const char *);

   // destructor : free everything
   ~Vector()        { if (vCapacity) delete [] data; }

   // assignment operator
   Vector<T>& operator=(const Vector<T> &rhs) throw (const char *);

   // square bracket operator
   T& operator [](int index) throw (const char *);
   const T& operator [](int index) const throw (const char *);

   // is the container currently empty
   bool empty() const  { return numItems == 0;         }

   // remove all the items from the container
```

```
   void clear()        { numItems = 0;                }

   // how many items are currently in the container?
   int size() const    { return numItems;             }

   // add an item to the container
   void insert(const T & t) throw (const char *);
   void push_back(const T & t) throw (const char *);

   // return the capacity
   int capacity() const { return vCapacity; }

   // return an iterator to the beginning of the list
   VectorIterator <T> begin() { return VectorIterator<T>(data); }

   // return an iterator to the end of the list
   VectorIterator <T> end() { return VectorIterator<T>(data + numItems);}

   // return an iterator to the beginning of the list
   VectorConstIterator <T> cbegin() const
         { return VectorConstIterator<T>(data); }

   // return an iterator to the end of the list
   VectorConstIterator <T> cend() const
         { return VectorConstIterator<T>(data + numItems);}

private:
   T * data;          // dynamically allocated array of T
   int numItems;      // how many items are currently in the Vector?
   int vCapacity;      // how many items can I put on the Vector before full?
};

/************************************************
 * CONTAINER ITERATOR
 * An iterator through Vector
 ************************************************/
template <class T>
class VectorIterator
{
  public:
   // default constructor
  VectorIterator() : p(NULL) {}

   // initialize to direct p to some item
  VectorIterator(T * p) : p(p) {}

   // copy constructor
   VectorIterator(const VectorIterator & rhs) { *this = rhs; }

   // assignment operator
   VectorIterator & operator = (const VectorIterator & rhs)
   {
      this->p = rhs.p;
      return *this;
   }

   // not equals operator
   bool operator != (const VectorIterator & rhs) const
   {
      return rhs.p != this->p;
   }

   // dereference operator
   T & operator * ()
   {
```

```
      return *p;
   }

   // prefix increment
   VectorIterator <T> & operator ++ ()
   {
      p++;
      return *this;
   }

   // postfix increment
   VectorIterator <T> operator++(int postfix)
   {
      VectorIterator tmp(*this);
      p++;
      return tmp;
   }

   // decrement operators
   VectorIterator <T> & operator -- ()
   {
      p--;
      return *this;
   }

   VectorIterator <T> operator--(int decrement)
   {
      VectorIterator tmp(*this);
      p--;
      return tmp;
   }

  private:
   T * p;
};

/************************************************
 * CONST CONTAINER ITERATOR
 * An const iterator through Vector
 ************************************************/
template <class T>
class VectorConstIterator
{
  public:
   // default constructor
  VectorConstIterator() : p(NULL) {}

   // initialize to direct p to some item
  VectorConstIterator(T * p) : p(p) {}

   // copy constructor
   VectorConstIterator(const VectorConstIterator & rhs) { *this = rhs; }

   // assignment operator
   VectorConstIterator & operator = (const VectorConstIterator & rhs)
   {
      this->p = rhs.p;
      return *this;
   }

   // not equals operator
   bool operator != (const VectorConstIterator & rhs) const
   {
      return rhs.p != this->p;
   }
```

```
   // dereference operator
   T & operator * ()
   {
      return *p;
   }

   // prefix increment
   VectorConstIterator <T> & operator ++ ()
   {
      p++;
      return *this;
   }

   // postfix increment
   VectorConstIterator <T> operator++(int postfix)
   {
      VectorConstIterator tmp(*this);
      p++;
      return tmp;
   }

   // decrement operators
   VectorConstIterator <T> & operator -- ()
   {
      p--;
      return *this;
   }

   VectorConstIterator <T> operator--(int decrement)
   {
      VectorConstIterator tmp(*this);
      p--;
      return tmp;
   }

  private:
   T * p;
};

/*****************************************
 * CONTAINER :: COPY CONSTRUCTOR
 *****************************************/
template <class T>
Vector <T> :: Vector(const Vector <T> & rhs) throw (const char *)
{
   assert(rhs.vCapacity >= 0);

   // do nothing if there is nothing to do
   if (rhs.vCapacity == 0)
   {
      vCapacity = numItems = 0;
      data = NULL;
      return;
   }

   // attempt to allocate
   try
   {
      data = new T[rhs.vCapacity];
   }
   catch (std::bad_alloc)
   {
      throw "ERROR: Unable to allocate buffer";
   }
```

```
   // copy over the capacity and size
   assert(rhs.numItems >= 0 && rhs.numItems <= rhs.vCapacity);
   vCapacity = rhs.vCapacity;
   numItems = rhs.numItems;

   // copy the items over one at a time using the assignment operator
   for (int i = 0; i < numItems; i++)
      data[i] = rhs.data[i];

   // the rest needs to be filled with the default value for T
   for (int i = numItems; i < vCapacity; i++)
      data[i] = T();
}

/*********************************************
 * CONTAINER : NON-DEFAULT CONSTRUCTOR
 * Preallocate the container to "vCapacity"
 *********************************************/
template <class T>
Vector <T> :: Vector(int vCapacity) throw (const char *)
{
   assert(vCapacity >= 0);

   // do nothing if there is nothing to do
   if (vCapacity == 0)
   {
      this->vCapacity = this->numItems = 0;
      this->data = NULL;
      return;
   }

   // attempt to allocate
   try
   {
      data = new T[vCapacity];
   }
   catch (std::bad_alloc)
   {
      throw "ERROR: Unable to allocate buffer";
   }


   // copy over the stuff
   this->vCapacity = vCapacity;
   this->numItems = 0;

   // initialize the container by calling the default constructor
   for (int i = 0; i < vCapacity; i++)
      data[i] = T();
}

/*********************************************
 * ASSIGNMENT OPERATOR
 *********************************************/
template <class T>
Vector <T>& Vector<T> :: operator=(const Vector<T> &rhs) throw (const char *)
{
   assert(rhs.vCapacity >= 0);

   // do nothing if there is nothing to do
   if (rhs.vCapacity == 0)
   {
      this->vCapacity = this->numItems = 0;
      this->data = NULL;
```

```
      return *this;
   }

   // attempt to allocate
   try
   {
      this->data = new T[rhs.vCapacity];
   }
   catch (std::bad_alloc)
   {
      throw "ERROR: Unable to allocate buffer";
   }

   // copy over the capacity and size
   assert(rhs.numItems >= 0 && rhs.numItems <= rhs.vCapacity);
   this->vCapacity = rhs.vCapacity;
   this->numItems = rhs.numItems;

   // copy the items over one at a time using the assignment operator
   for (int i = 0; i < this->numItems; i++)
      this->data[i] = rhs.data[i];

   // the rest needs to be filled with the default value for T
   for (int i = this->numItems; i < this->vCapacity; i++)
      this->data[i] = T();

   return *this;
}

/*************************************************
 * OPERATOR []
 *************************************************/
template <class T>
T& Vector<T> :: operator[](int index) throw (const char *)
{
   if (index < 0 || index >= numItems)
   {
      throw "ERROR: index out of bounds";
   }

   return data[index];
}

template <class T>
const T& Vector<T> :: operator[](int index) const throw (const char *)
{
   if (index < 0 || index >= numItems)
   {
      throw "ERROR: index out of bounds";
   }

   return data[index];
}

/*************************************************
 * CONTAINER :: INSERT
 * Insert an item on the end of the container
 *************************************************/
template <class T>
void Vector <T> :: insert(const T & t) throw (const char *)
{
   // do we have space?
   if (vCapacity == 0 || vCapacity == numItems)
      throw "ERROR: Insufficient space";
```

```
   // add an item to the end
   data[numItems++] = t;
}

/*****************************************************
 * PUSH BACK
 * Extends the size of the container if necessary, before inserting item
 *****************************************************/
template <class T>
void Vector <T> :: push_back(const T & t) throw (const char *)
{
   T* newData;

   if (numItems >= vCapacity)
   {
      if (vCapacity == 0)
      {
         vCapacity = 1;
      }
      else
      {
         vCapacity *= 2;
      }
      newData = new T[vCapacity];
      for (int idx = 0; idx < numItems; ++idx)
      {
         newData[idx] = data[idx];
      }
      delete [] data;
      data = newData;
   }


   insert(t);
}

#endif // CONTAINER_H
```