makefile

```
################################################################
# Program:
#     Lesson 06, LIST
#     Brother Helfrich, CS235
# Author:
#     <your name here>
# Summary:
#     <put a description here>
# Time:
#     <how long did it take to complete this program>?
################################################################

################################################################
# The main rule
################################################################
a.out: list.h lesson06.o fibonacci.o
       g++ -g -o a.out lesson06.o fibonacci.o
       tar -cf lesson06.tar *.h *.cpp makefile

################################################################
# The individual components
#      lesson06.o     : the driver program
#      fibonacci.o    : the logic for the fibonacci-generating function
#      <anything else?>
################################################################
lesson06.o: list.h lesson06.cpp
       g++ -g -c lesson06.cpp

fibonacci.o: fibonacci.h fibonacci.cpp list.h
       g++ -g -c fibonacci.cpp
```

Commented [HJ1]: !!!! Why throw away perfectly good points?

node.h

```
/*************************************************************************
 * Header:
 *    NODE
 * Summary:
 *    This contains the class for our Node. It also contains the different
 *    stand alone functions allowing us to insert into a list, copy, find and
 *    free the data.
 * Author
 *    David Lambertson
 *************************************************************************/


#ifndef NODE_H
#define NODE_H

#include <iostream>
#include <cassert>

/*********************
 *Node class to create nodes
 * to be used within a linked list
 *********************/
template <class T>
class Node
{
  public:
   T data;
   Node<T> * pNext;
   Node<T> * pPrev;

   Node() : data(), pNext(NULL), pPrev(NULL) {}
   Node(T data)
   {
      this->data = data;
```

```
      pNext = NULL;
      pPrev = NULL;
   }
};

/***************************
 * allows the program to create copy
 * a list of Nodes
 ***************************/
template <class T>
Node<T> * copy(Node<T> * p)
{
   Node<T> * pHead = NULL;
   Node<T> * pNew = new Node<T>;

   for (; p; p = p->pNext)
   {
      Node<T> * tmp = new Node<T>;
      tmp->data=p->data;
      if (pHead == NULL)
      {
         tmp->pNext = NULL;
         tmp->pPrev = NULL;
         pNew = tmp;
         pHead = pNew;
      }
      else
      {
         pNew->pNext = tmp;
         tmp->pPrev = pNew;
         pNew = tmp;
         tmp->pNext = NULL;
      }

   }

   return pHead;
}

/***************************
 * lets us insert into the list
 ***************************/
template <class T>
void insert(T value, Node<T> * & p)
{

   if (p == NULL)
   {
      insert(value, p, true);
   }
   else
   {
      Node<T> * pNew = new Node<T>;
      pNew->data = value;
      pNew->pNext = p->pNext;
      pNew->pPrev = p;
      if (p->pNext != NULL)
         p->pNext->pPrev = pNew;
      p->pNext = pNew;

   }
}

/***************************
 *insert when at front of head
 ***************************/
template <class T>
void insert(T value, Node<T> * & p, bool head)
{
   assert(head);
   Node<T> * pNew = new Node<T>;
   pNew->data = value;
   pNew->pNext = p;
   if (p != NULL)
      p->pPrev = pNew;
   p = pNew;
   pNew->pPrev = NULL;
}
```

Commented [HJ2]: Can be done in the initialization section.

Commented [HJ3]: Should be a member function in List

Commented [HJ4]: Same here. Should be a member function.

```
/*****************************
 *finds if the value given is within the list we have
 ************************************/
template <class T>
Node<T> * find(Node<T> * & pHead, T value)
{

    if (pHead == NULL)
   {
      return NULL;
   }
   Node<T> * found = new Node<T>;
   for (Node<T> * p = pHead; p != NULL; p = p->pNext)
   {
      if (p->data == value)
      {
         found = p;
         break;
      }
      else
         found = NULL;

   }
   return found;
}

/*******************************
 *frees the data taken up by the list
 *******************************/
template <class T>
void freeData(Node<T> * & pHead)
{
   Node<T> * p;
   for (p = pHead; p; p = p->pNext)
   {
      Node<T> * tmp = p;
      delete tmp;
   }
   pHead = NULL;
}

/*****************************************************
 *overloaded insertion operator allowing us to make displaying more easy
 *****************************************************/
template <class T>
std::ostream& operator << (std::ostream& out, const Node<T> * p)
{
   for (const Node<T> * pNew = p; pNew; pNew = pNew->pNext)
   {
      if (pNew->pNext != NULL)
         out << pNew->data << ", ";
      else
         out << pNew->data;
   }

   return out;
}

#endif
```

list.h

```
/***********************************************************************
 * Header:
 *    List
 * Summary:
 *    This is the List and ListIterator classes which are needed
 *    to create a list that can be inserted, removed, and cleared
 *    in constant time rather than a longer computational time.
 * Author:
 *    Derek Calkins
 * Hardest Part:
 *    Finding out how the ListIterator is supposed to use the methods
 *    and member variables of the List class.
 * Time:
 *    Expected: Approx. 5  hours
 *    Actual:   Approx. 10 hours
 ***********************************************************************/
```

```
#ifndef LIST_H
#define LIST_H

#include <iostream>
#include <cassert>
#include "node.h"

template <class T>
class ListIterator;

/*********************
 * List class to which creates Nodes
 * which are used to create a list
 *********************/
template <class T>
class List
{
public:
   //default constructor
  List() : pHead(NULL), pTail(NULL) {}

   //non-default constructor
   List(T data)
   {
      pHead = NULL;
      pTail = NULL;
   }

   //copy constructor
   List(const List & rhs)
   {
      this->pHead = rhs.pHead;
      this->pTail = rhs.pTail;
   }

   //overloaded assignment operator
   List<T> operator = (const List<T> & rhs);

   //checks if the list is empty
   bool empty()
   {
      if (pHead == NULL)
         return true;
      else
         return false;
   }

   //removes all nodes and pointers
   void clear();

   //adds the value to the end of the list
   void push_back(T value);

   //adds a value to the beginning of the list
   void push_front(T value);

   //returns the value at the beginning of the list
   T & front() throw (const char *)
   {
      if(empty())
         throw "ERROR: unable to access data from empty list";
      return pHead->data;
   }

   //returns the value at the end of the list
   T & back() throw (const char *)
   {
      if(empty())
         throw "ERROR: unable to access data from empty list";
      return pTail->data;
   }

   //inserts a value into the specified spot in the list
   void insert(ListIterator <T> & it, const T & value);

   //removes a node at the specified spot in the list
   void remove(ListIterator <T> & it) throw (const char *);
```

Commented [HJ5]: Should be a const method.

Commented [HJ6]: Shoud lthrow!

Commented [HJ7]: Can throw

```
   //returns the pointer to the first node in the list
   ListIterator<T> begin()
   {
      if (pHead == NULL)
         return NULL;
      return pHead;
   }

   //returns the pointer to past the last element of the list
   ListIterator<T> end()
   {
      if (pTail == NULL)
         return NULL;
      return pTail->pNext;
   }

   //returns the pointer to the first element of the reverse list
   ListIterator<T> rbegin()
   {
      //if (pTail == NULL)
      //   return NULL;
      return pTail;
   }

   //returns the pointer to the past the last element of the reverse list
   ListIterator<T> rend()
   {
      if(pHead == NULL)
         return NULL;
      return pHead->pPrev;
   }

private:
   Node<T> * pHead;  //Node to the beginning of the list
   Node<T> * pTail;  //Node to the end of the list
};

/***********************************
 * ListIterator class which has one Node
 * to determine where an item needs to be
 * inserted or removed from a List
 ***********************************/
template <class T>
class ListIterator
{
public:
   //Default Constructor
  ListIterator() : p(NULL) {}

   //Non Default Constructor
  ListIterator(Node<T> * p) : p(p) {}

   //copy Constructor
   ListIterator(const ListIterator & rhs) { this->p = rhs.p; }

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

   bool operator == (const ListIterator & rhs)
   {
      return this->p == rhs.p;
   }

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

   //overloaded by-reference operator
   T & operator * ()
   {
      return p->data;
   }
```

Commented [HJ8]: This is not an iterator.

Commented [HJ9]: Need to return an iterator.

Commented [HJ10]: Good.

Commented [HJ11]: Goo.d

```
   //overloaded prefix add one operator
   ListIterator <T> & operator ++ ()
   {
      p = p->pNext;
      return *this;
   }

   //overloaded postfix add one operator
   ListIterator <T> operator ++ (int postfix)
   {
      ListIterator tmp(*this);
      p = p->pNext;
      return tmp;
   }

   //overloaded prefix subtract one operator
   ListIterator <T> & operator -- ()
   {
      p = p->pPrev;
      return *this;
   }

   //overloaded postfix subtract one operator
   ListIterator <T> operator -- (int postfix)
   {
      ListIterator tmp(*this);
      p = p->pPrev;
      return *this;
   }

   //ListIterator Insert uses methods and variables from List
   template <class U>
   friend List<T> insert(ListIterator <U> & it,
                         const T & value);

   //ListIterator Remove uses methods and variables from List
   template <class U>
   friend List<T> remove(ListIterator <U> & it) throw (const char *);


   Node<T> * p; //Node where the ListIterator is pointing
};

/***************************
 * OVERLOADED ASSIGNMENT OPERATOR
 * Allows the program to copy
 * one list into another list
 ***************************/
template <class T>
List<T> List<T> :: operator = (const List<T> & rhs)
{
   Node<T> * pNewList = NULL;
   Node<T> * pNew = new Node<T>;
//   if(std::bad_alloc())
//      throw "Unable to allocate a new node for a list";

   for (; pNewList; pNewList = pNewList->pNext)
   {
      Node<T> * tmp = new List<T>;
      tmp->data=rhs->data;
      if (pNewList == NULL)
      {
         tmp->pNext = NULL;
         tmp->pPrev = NULL;
         pNew = tmp;
         pHead = pNew;
         pTail = pNew;
      }
      else
      {
         pNew->pNext = tmp;
         tmp->pPrev = pNew;
         pNew = tmp;
         tmp->pNext = NULL;
         pTail = pNew;
      }

   }
```

Commented [HJ12]: Should be private.

Commented [HJ13]: It would be much easier if you just call push_back()

```
   std::cout << empty() << std::endl;

   return pNewList;
}

/*****************************
 * INSERT
 * Lets us insert a Node into the middle
 * of the List
 *****************************/
template <class T>
void List<T> :: insert(ListIterator<T> & it, const T & value)
{
   Node<T> * pNew = new Node<T>;
//   catch(std::bad_alloc&)
//      throw "Unable  to allocate a new node for a list";
   pNew->data = value;

   if(it.p == NULL)
   {
      push_back(value);
   }
   else if(it.p->pPrev == NULL)
   {
      push_front(value);
   }
   else
   {
      pNew->pNext = it.p;
      pNew->pPrev = it.p->pPrev;
      it.p->pPrev->pNext = pNew;
      it.p->pPrev = pNew;
   }
}

/*****************************
 * REMOVE
 * Takes the iterator to the Node we want to remove
 * and depending where that is, deletes the Node
 * and fixes the other Nodes so we don't lose the List.
 *************************************/
template <class T>
void List<T> :: remove(ListIterator<T> & it) throw (const char *)
{
   if(it.p == NULL)
      throw "unable to remove from an invalid location in list";
   else if(it.p->pNext == NULL)
   {
      pTail = it.p->pPrev;
      it.p->pPrev->pNext = NULL;
      delete [] it.p;
   }
   else if(it.p->pPrev == NULL)
   {
      pHead = it.p->pNext;
      it.p->pNext->pPrev = NULL;
      delete [] it.p;
   }
   else
   {
      it.p->pNext->pPrev = it.p->pPrev;
      it.p->pPrev->pNext = it.p->pNext;
      delete [] it.p;
   }
}

/*********************************
 * CLEAR
 * Frees the data by deleting all the
 * Nodes in the list
 *********************************/
template <class T>
void List<T> :: clear()
{
   Node<T> * p1;
   for (p1 = pHead; p1; p1 = p1->pNext)
   {
      Node<T> * tmp = p1;
      delete tmp;
```

Commented [HJ14]: ?? Why?

```
   }
   pHead = NULL;
   pTail = NULL;
}

template <class T>
void List<T> :: push_back(T value)
{
   //Create a new Node
   Node<T> * pNew = new Node<T>;
//   if(bad_alloc())
//      throw "Unable to allocate a new node for a list";

   pNew->data = value;

   //if we have nothing in the list just push_front
   if(pTail == NULL)
      push_front(value);
   //if we already have something in the list add at the end
   else
   {
      pNew->pNext = pTail->pNext;
      pNew->pPrev = pTail;
      if (pTail->pNext != NULL)
         pTail->pNext->pPrev = pNew;
      pTail->pNext = pNew;
      pTail = pNew;
      if (pNew->pPrev == NULL)
         pHead = pTail;
   }
}

//adds a value to the beginning of the list
template <class T>
void List<T> :: push_front(T value)
{
   //create a new Node
   Node<T> * pNew = new Node<T>;
//   if(std::bad_alloc())
//      throw "Unable to allocate a new node for a list";
   pNew->data = value;

   //adds at the beginning of the list
   pNew->pNext = pHead;
   if (pHead != NULL)
      pHead->pPrev = pNew;
   pHead = pNew;
   pNew->pPrev = NULL;
   pHead = pNew;
   if (pNew->pNext == NULL)
      pTail = pHead;
}

#endif // LIST_H
```

fibonacci.h

```
/**********************************************************************
 * Header:
 *    FIBONACCI
 * Summary:
 *    This will contain just the prototype for fibonacci(). You may
 *    want to put other class definitions here as well. It includes my Whole
 *    Number class for dealing with big numbers.
 * Author
 *    <your names here>
 **********************************************************************/

#ifndef FIBONACCI_H
#define FIBONACCI_H

#include "list.h"

#include <iostream>
#include <list>
#include <iomanip>
```

```
// the interactive fibonacci program
void fibonacci();

class WholeNumber
{
  public:

   //Default constructor
  WholeNumber() : fibo() {}

   //Destructor
   ~WholeNumber()
   {
      //fibo.clear();
   }


   //Non-default Destructor
   WholeNumber(unsigned int i)
   {
      fibo.push_back(i);
   }

   /***************************************
    *this method allows me to clear whats in my list
    ***************************************/
   void reset() { fibo.clear(); fibo.push_back(1); }

   //Prototype for my += operator
   void operator += (WholeNumber & rhs);

   //overloading the
   friend std::ostream& operator <<(std::ostream& out, WholeNumber number);


  private:
    std::list<int> fibo;    //my list for my Big numbers
 };


#endif // FIBONACCI_H
```

Commented [HJ15]: Need a comment block.

Commented [HJ16]: What if this is bigger than 999?

Commented [HJ17]: Just assign to zero.

Commented [HJ18]: Good. It would be better if it was your own List class.

lesson06.cpp

```
/***********************************************************************
* Program:
*    Lesson 06, LIST
*    Brother Helfrich, CS 235
* Author:
*    Br. Helfrich
* Summary:
*    This is a driver program to exercise the List class.  When you
*    submit your program, this should not be changed in any way.  That being
*    said, you may need to modify this once or twice to get it to work.
***********************************************************************/

#include <iostream>     // for CIN and COUT
#include <iomanip>      // for SETW
#include <string>       // for the String class
#include "list.h"       // your List class should be in list.h
#include "fibonacci.h"  // your fibonacci() function
using namespace std;


// prototypes for our four test functions
void testSimple();
void testPush();
void testIterate();
void testInsertRemove();

// To get your program to compile, you might need to comment out a few
// of these. The idea is to help you avoid too many compile errors at once.
// I suggest first commenting out all of these tests, then try to use only
// TEST1.  Then, when TEST1 works, try TEST2 and so on.
#define TEST1   // for testSimple()
#define TEST2   // for testPush()
#define TEST3   // for testIterate()
```

```
#define TEST4   // for testInsertRemove()

/*****************************************************************
 * MAIN
 * This is just a simple menu to launch a collection of tests
 *****************************************************************/
int main()
{
   // menu
   cout << "Select the test you want to run:\n";
   cout << "\t0. Fibonacci\n";
   cout << "\t1. Just create and destroy a List\n";
   cout << "\t2. The above plus push items onto the List\n";
   cout << "\t3. The above plus iterate through the List\n";
   cout << "\t4. The above plus insert and remove items from the list\n";

   // select
   int choice;
   cout << "> ";
   cin  >> choice;
   switch (choice)
   {
      case 0:
         fibonacci();
         break;
      case 1:
         testSimple();
         cout << "Test 1 complete\n";
         break;
      case 2:
         testPush();
         cout << "Test 2 complete\n";
         break;
      case 3:
         testIterate();
         cout << "Test 3 complete\n";
         break;
      case 4:
         testInsertRemove();
         cout << "Test 4 complete\n";
         break;
      default:
         cout << "Unrecognized command, exiting...\n";
   }

   return 0;
}

/*****************************************
 * TEST SIMPLE
 * Very simple test for a List: create and destroy
 *****************************************/
void testSimple()
{
#ifdef TEST1
   cout.setf(ios::fixed | ios::showpoint);
   cout.precision(5);

   // Test1: a bool List with default constructor
   cout << "Create a bool List using the default constructor\n";
   List <bool> l1;
   cout << "\tEmpty? " << (l1.empty() ? "Yes" : "No") << endl;

   // Test2: double List and add one element
   cout << "Create a double List and add one element: 3.14159\n";
   List <double> l2;
   l2.push_back(3.14159);
   cout << "\tEmpty? " << (l2.empty() ? "Yes" : "No") << endl;
   cout << "\tFront: " << l2.front()                  << endl;
   cout << "\tBack:  " << l2.back()                   << endl;

   {
      // Test3: copy the double List
      cout << "Copy the double List using the copy-constructor\n";
      List <double> l3(l2);
      cout << "\tEmpty? " << (l3.empty() ? "Yes" : "No") << endl;
      cout << "\tFront: " << l3.front()                  << endl;
      cout << "\tBack:  " << l3.back()                   << endl;
   }
```

```
   cout << "\tDestroying the third List\n";
#endif //TEST1
}

/*****************************************
 * TEST PUSH
 * Add a whole bunch of items to the List.  This will
 * test the push_back() and push_front() algorithm
 *****************************************/
void testPush()
{
#ifdef TEST2
   // create
   cout << "Create an integer List with the default constructor\n";
   List <int> l;
   cout << "\tEmpty? " << (l.empty() ? "Yes" : "No") << endl;

   // test push_back
   cout << "Test push_back() by adding items to the back of the list\n";
   cout << "\tEnter integer values, type 0 when done\n";
   int value;
   do
   {
      cout << "\t> ";
      cin  >> value;
      if (value)
      {
         l.push_back(value);

//         for (ListIterator<int> it = l.begin(); it != l.end(); ++it)
//            cout << *it << endl;

         // display the front and the back
         cout << "\t\tFront: " << l.front()
              << "    Back: "  << l.back()
              << endl;
      }
   }
   while (value);

   // test empty
   cout << "Test clear() to remove all the items\n";
   l.clear();
   cout << "\tEmpty? " << (l.empty() ? "Yes" : "No") << endl;

   // test push_front
   cout << "Test push_front() by adding items to the front of the list\n";
   cout << "\tEnter integer values, type 0 when done\n";
   do
   {
      cout << "\t> ";
      cin  >> value;
      if (value)
      {
         l.push_front(value);

//         for (ListIterator<int> it = l.begin(); it!= l.end(); ++it)
//            cout << *it << endl;


         // display the front and the back
         cout << "\t\tFront: " << l.front()
              << "    Back: "  << l.back()
              << endl;
      }
   }
   while (value);
#endif // TEST2
}

/*****************************************
 * TEST ITERATE
 * We will test the iterators. We will go through the
 * list forwards and backwards
 *****************************************/
void testIterate()
{
#ifdef TEST3
   // create
```

```
   cout << "Create a string List with the default constructor\n";
   List <string> l;

   // instructions
   cout << "Instructions:\n"
        << "\t+ dog   pushes dog onto the front\n"
        << "\t- cat   pushes cat onto the back\n"
        << "\t#       displays the contents of the list\n"
        << "\t*       clear the list\n"
        << "\t!       quit\n";

   char command;
   string text;
   do
   {
      cout << "> ";
      cin  >> command;

      try
      {
         switch (command)
         {
            case '+':
               cin >> text;
               l.push_front(text);
               break;
            case '-':
               cin >> text;
               l.push_back(text);
               break;
            case '#':
            {
               ListIterator <string> it;
               cout << "\tForwards: ";
               for (it = l.begin(); it != l.end(); ++it)
                  cout << ' ' << *it;
               cout << endl;

               cout << "\tBackwards:";
               for (it = l.rbegin(); it != l.rend(); --it)
                  cout << ' ' << *it;
               cout << endl;
               break;
            }
            case '*':
               l.clear();
               break;
            case '!':
               break;
            default:
               cout << "Unknown command\n";
               cin.ignore(256, '\n');
         }
      }
      catch (const char * e)
      {
         cout << '\t' << e << endl;
      }
   }
   while (command != '!');
#endif // TEST3
}

/*****************************************
 * TEST INSERT REMOVE
 * We will insert items in a list from the location
 * specified by the iterator, and remove items from
 * the list from the given iterator
 *****************************************/
void testInsertRemove()
{
#ifdef TEST4
   // first, fill the list
   List <char> l;
   for (char letter = 'a'; letter <= 'm'; letter++)
      l.push_back(letter);

   // instructions
   cout << "Instructions:\n"
```

```
         << "\t+ 3 A  put 'A' after the 3rd item in the list\n"
         << "\t- 4    remove the fourth item from the list\n"
         << "\t!       quit\n";

   char command;
   do
   {
      ListIterator <char> it;
      int index = 0;
      char letter;

      // display the list
      for (it = l.begin(); it != l.end(); ++it)
         cout << setw(3) << index++;
      cout << endl;
      for (it = l.begin(); it != l.end(); ++it)
         cout << setw(3) << *it;

      // prompt for the next command
      cout << "\n> ";
      cin  >> command;

      try
      {
         switch (command)
         {
            case '+':
               cin >> index >> letter;
               it = l.begin();
               while (index-- > 0)
                  ++it;
               l.insert(it, letter);
               break;
            case '-':
               cin >> index;
               it = l.begin();
               while (index-- > 0)
                  ++it;
               l.remove(it);
               break;
            case '!':
               break;
            default:
               cout << "Unknown command\n";
               break;
         }

         // error recovery: unexpected input
         if (cin.fail())
         {
            cin.clear();
            cin.ignore(256, '\n');
         }

      }
      // error recovery: thrown exception
      catch (const char * e)
      {
         cout << '\t' << e << endl;
      }
   }
   while (command != '!');
#endif // TEST4
}
```

fibonacci.cpp

```
/**********************************************************************
 * Implementation:
 *    FIBONACCI
 * Summary:
 *    This will contain the implementation for fibonacci() as well as any
 *    other function or class implementations you may need
 * Author
 *    David Lambertson
 **********************************************************************/

#include <iostream>
```

```
#include "fibonacci.h"   // for fibonacci() prototype
#include "list.h"        // for LIST
using namespace std;


/*********************************************
 * FIBONACCI
 * The interactive function allowing the user to
 * display Fibonacci numbers
 *********************************************/
void fibonacci()
{
   // show the first serveral Fibonacci numbers
   int number;
   cout << "How many Fibonacci numbers would you like to see? ";
   cin  >> number;

   // your code to display the first <number> Fibonacci numbers


   WholeNumber number1(1);
   WholeNumber number2(1);
   cout << "\t" << number1 << endl;
   cout << "\t" << number2 << endl;   //create two WholeNumbers and display them for the first two fibonacci number
s

   for (int i = 2; i < number; i++) //keep going until I get to where they want me.
   {
      if (i%2 == 0) //even
      {
         number1 += number2;
         cout << "\t" << number1 << endl;
      }
      else       //odd
      {
         number2 += number1;
         cout << "\t" << number2 << endl;
      }
   }

   // prompt for a single large Fibonacci
   cout << "Which Fibonacci number would you like to display? ";
   cin  >> number;

   number1.reset();  //resets my WholeNumbers back to the beginning
   number2.reset();

   for (int i = 2; i < number; i++)
   {
      if (i%2 == 0)
      {
         number1 += number2;
      }
      else
      {
         number2 += number1;
      }
   }

   if (number%2 == 0) //display only the number they want.
      cout << "\t" << number2 << endl;
   else
      cout << "\t" << number1 << endl;
   // your code to display the <number>th Fibonacci number
}

/***********************************
 * overloaded insertion opperator to display my WholeNumbers
 ***********************************/
std::ostream& operator <<(std::ostream& out, WholeNumber number)
{
   for (list<int>::const_reverse_iterator it = number.fibo.rbegin();
        it != number.fibo.rend(); ++it)
   {
      if (it != --number.fibo.rend())//do this until I am at end of list
      {
         if (it == number.fibo.rbegin())
            out << *it << ',';
         else
```

Commented [HJ19]: Instead try: number1 = 0;

Commented [HJ20]: Nicely done.

```
            out << setfill('0') << setw(3) << *it << ',';
      }
      else // do this when I am at end of list
      {
         if (it == number.fibo.rbegin())
            out << *it;
         else
            out << setfill('0') << setw(3) << *it;
      }
   }

   return out;
}

/*****************************************
 * overloaded += operator to allow me to go through the fibonacci numbers
 *****************************************/
void WholeNumber :: operator += (WholeNumber & rhs)
{
   list<int>::iterator it2 = rhs.fibo.begin();
   list<int>::iterator it = fibo.begin();

   int carry = 0;
   for (; it2 != rhs.fibo.end(); ++it2)
   {

      int x=0, y=0;
      if (it != fibo.end())
         x = *it;
      else
         x = 0; //fibo

      if (it2 != rhs.fibo.end())
         y = *it2; //rhs fibo
      else
         y = 0;

      int a = x + y + carry;

      carry = a / 1000;

      if (it == fibo.end())
      {
         fibo.push_back(a);
         break;
      }
      *it = (a%1000);
      ++it;
   }

   if (carry)
      fibo.push_back(carry);
}
```

Commented [HJ21]: This should be outside the loop.

Commented [HJ22]: Horrible variable names. The style is really bad here.

Test Bed Results

```
cs235d.out:

Started program
   > Select the test you want to run:
   >    0. Fibonacci
   >    1. Just create and destroy a List
   >    2. The above plus push items onto the List
   >    3. The above plus iterate through the List
   >    4. The above plus insert and remove items from the list
   > > 1
   > Create a bool List using the default constructor
   >    Empty? Yes
   > Create a double List and add one element: 3.14159
   >    Empty? No
   >    Front: 3.14159
   >    Back:  3.14159
   > Copy the double List using the copy-constructor
   >    Empty? No
   >    Front: 3.14159
   >    Back:  3.14159
   >    Destroying the third List
```

```
   > Test 1 complete
Program terminated successfully

Started program
   > Select the test you want to run:
   >    0. Fibonacci
   >    1. Just create and destroy a List
   >    2. The above plus push items onto the List
   >    3. The above plus iterate through the List
   >    4. The above plus insert and remove items from the list
   > > 2
   > Create an integer List with the default constructor
   >    Empty? Yes
   > Test push_back() by adding items to the back of the list
   >    Enter integer values, type 0 when done
   >    > 2
   >       Front: 2    Back: 2
   >    > 4
   >       Front: 2    Back: 4
   >    > 6
   >       Front: 2    Back: 6
   >    > 8
   >       Front: 2    Back: 8
   >    > 0
   > Test clear() to remove all the items
   >    Empty? Yes
   > Test push_front() by adding items to the front of the list
   >    Enter integer values, type 0 when done
   >    > 1
   >       Front: 1    Back: 1
   >    > 3
   >       Front: 3    Back: 1
   >    > 5
   >       Front: 5    Back: 1
   >    > 7
   >       Front: 7    Back: 1
   >    > 0
   > Test 2 complete
Program terminated successfully

Started program
   > Select the test you want to run:
   >    0. Fibonacci
   >    1. Just create and destroy a List
   >    2. The above plus push items onto the List
   >    3. The above plus iterate through the List
   >    4. The above plus insert and remove items from the list
   > > 3
   > Create a string List with the default constructor
   > Instructions:
   >    + dog   pushes dog onto the front
   >    - cat   pushes cat onto the back
   >    #       displays the contents of the list
   >    *       clear the list
   >    !       quit
   > > +three
   > > +four
   > > +five
   > > -two
   > > -one
   > > -zero
   > > #
   >    Forwards:  five four three two one zero
   >    Backwards: zero one two three four five
   > > *
   > > #
   >    Forwards:
   >    Backwards:
   > > +front
   > > -back
   > > #
   >    Forwards:  front back
   >    Backwards: back front
   > > !
   > Test 3 complete
Program terminated successfully

Started program
   > Select the test you want to run:
```

```
   >    0. Fibonacci
   >    1. Just create and destroy a List
   >    2. The above plus push items onto the List
   >    3. The above plus iterate through the List
   >    4. The above plus insert and remove items from the list
   > > 4
   > Instructions:
   >    + 3 A  put 'A' after the 3rd item in the list
   >    - 4    remove the fourth item from the list
   >    !       quit
   >   0  1  2  3  4  5  6  7  8  9 10 11 12
   >   a  b  c  d  e  f  g  h  i  j  k  l  m
   > > -1
   >   0  1  2  3  4  5  6  7  8  9 10 11
   >   a  c  d  e  f  g  h  i  j  k  l  m
   > > -2
   >   0  1  2  3  4  5  6  7  8  9 10
   >   a  c  e  f  g  h  i  j  k  l  m
   > > -1
   >   0  1  2  3  4  5  6  7  8  9
   >   a  e  f  g  h  i  j  k  l  m
   > > +1B
   >   0  1  2  3  4  5  6  7  8  9 10
   >   a  B  e  f  g  h  i  j  k  l  m
   > > +2D
   >   0  1  2  3  4  5  6  7  8  9 10 11
   >   a  B  D  e  f  g  h  i  j  k  l  m
   > > +2C
   >   0  1  2  3  4  5  6  7  8  9 10 11 12
   >   a  B  C  D  e  f  g  h  i  j  k  l  m
   > > -12
   >   0  1  2  3  4  5  6  7  8  9 10 11
   >   a  B  C  D  e  f  g  h  i  j  k  l
   > > -0
   >   0  1  2  3  4  5  6  7  8  9 10
   >   B  C  D  e  f  g  h  i  j  k  l
   > > +0A
   >   0  1  2  3  4  5  6  7  8  9 10 11
   >   A  B  C  D  e  f  g  h  i  j  k  l
   > > +12M
   >   0  1  2  3  4  5  6  7  8  9 10 11 12
   >   A  B  C  D  e  f  g  h  i  j  k  l  M
   > > !
   > Test 4 complete
Program terminated successfully

Started program
   > Select the test you want to run:
   >    0. Fibonacci
   >    1. Just create and destroy a List
   >    2. The above plus push items onto the List
   >    3. The above plus iterate through the List
   >    4. The above plus insert and remove items from the list
   > > 0
   > How many Fibonacci numbers would you like to see? 10
   >    1
   >    1
   >    2
   >    3
   >    5
   >    8
   >    13
   >    21
   >    34
   >    55
   > Which Fibonacci number would you like to display? 500
   >    139,423,224,561,697,880,139,724,382,870,407,283,950,070,256,587,697,307,264,108,962,948,325,571,622,863,290
,691,557,658,876,222,521,294,125
Program terminated successfully

No Errors
```

Grading Criteria

Criteria	Exceptional 100%	Good 90%	Acceptable 70%	Developing 50%	Missing 0%	Weight	Score
List interface	The interfaces are perfectly specified with respect to const, pass-by-reference, etc.	lesson06.cpp compiles without modification	All of the methods in List match the problem definition	List has many of the same interfaces as the problem definition	The public methods in the List class do not resemble the problem definition	20	18
List Implementation	Passes all four List testBed tests	Passes three testBed tests	Passes two testBed tests	Passes one testBed test	Program fails to compile or does not pass any testBed tests	20	20
Whole Numbers	The WholeNumber class supports all the common operators perfectly	A WholeNumber class exists but does not implement any of the common operators	Able to perfectly handle large numbers without a WholeNumber class -or- a WholeNumber class exists but has one minor bug	An attempt was made to use the List class to represent large numbers	No attempt was made to handle large whole numbers	30	30
Fibonacci	The most efficient solution was found	Passes the Fibonacci testBed test	The code essentially works but with minor defects	Elements of the solution are present	The Fibonacci problem was not attempted	10	9
Code Quality	There is no obvious room for improvement	All the principles of encapsulation and modularization are honored	One function is written in a "backwards" way or could be improved	Two or more functions appears "thrown together."	The code appears to be written without any obvious forethought	10	7
Style	Great variable names, no errors, great comments	No obvious style errors	A few minor style errors: non-standard spacing, poor variable names, missing comments, etc.	Overly generic variable names, misleading comments, or other gross style errors	No knowledge of the BYU-I code style guidelines were demonstrated	10	7
Extra Credit	10% Implement a ListConstIterator	5% Extend the WholeNumber class to include subtraction	10% Extend the WholeNumber class to include the extraction operator	10% Extend the WholeNumber class to include multiplication		40	0
Total							91

Commented [HJ23]: A bit of this is sloppy. The score would be much higher if you spent a half hour cleaning up the code. I don't feel this is in a "finished" state.