07 : List

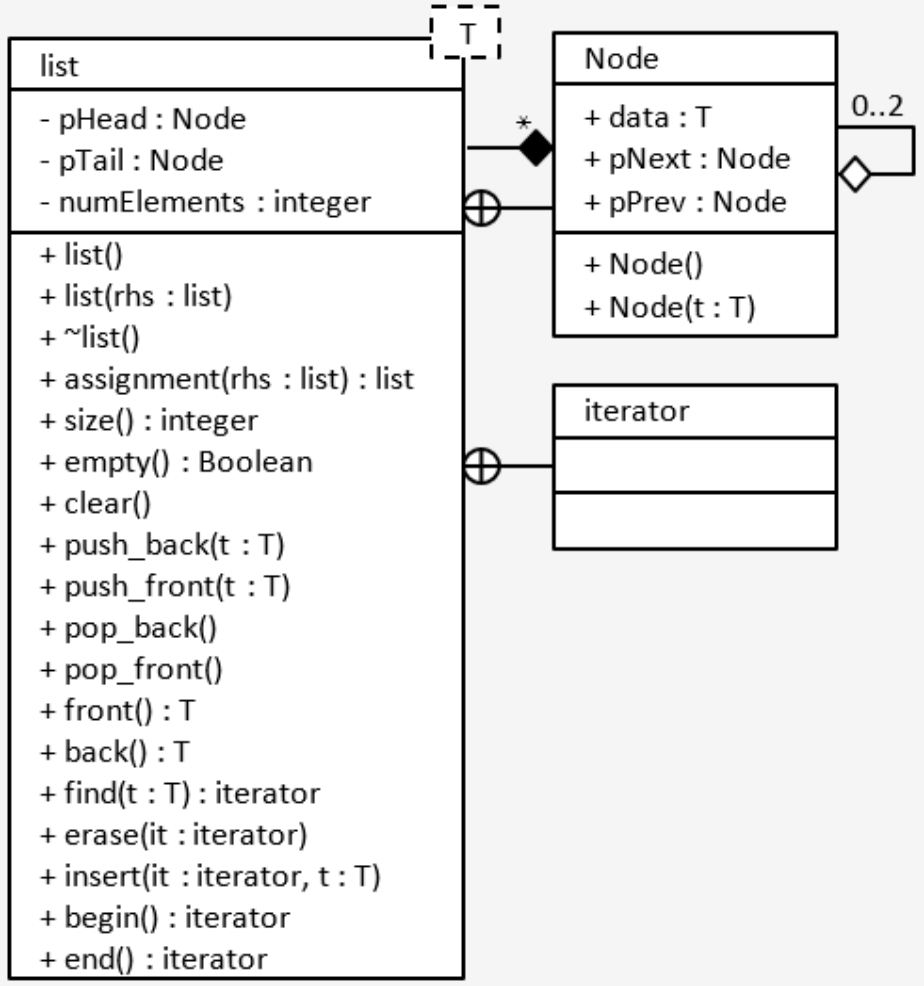
PONDER 07 : FIBONACCI

Due Saturday at 5:00 PM MST

The seventh programming assignment will be to create the list data structure and use it to implement a program to find very large Fibonacci numbers. Many students say this is the most difficult assignment to date. This will be accomplished using the Pair Programming technique.

List

Create a class encapsulating the notion of a list. This list will be a wrapper around the linked-list we created last week. In other words, it will honor all the common container interfaces without revealing to the client any of the internal details. We will create a `list` class that will work exactly like the `std::list` class. Of course, any data-type will need to be supported, so your class will be a template class. It will need to be defined in its own header file (`list.h`). The class name must be `list`.



Your class will need to support the following operations:

- **Constructors**: Default constructor (create an empty list) and the copy constructor. If allocation is not possible, the following error will be thrown:
  `ERROR: unable to allocate a new node for a list`

- **Destructor**: When finished, the class should delete all the allocated memory.
- **`operator=()`**: Removes all the elements in the current list and copies the contents from the right-hand side (`rhs`) onto the current list. In the case of an allocation error, the following c-string will be thrown:
  `ERROR: unable to allocate a new node for a list`
- **`empty()`**: Test whether the list is empty. This method takes no parameters and returns a Boolean value.
- **`size()`**: Returns the number of nodes in the list. There are no parameters and the return value is an integer.
- **`clear()`**: Empties the list of all elements. There are no parameters and no return value.
- **`push_back()`**: Adds an element to the back of the list. This method takes a single parameter (the element to be added to the end of the list) and has no return value. In the case of an allocation error, the following c-string exception will be thrown:
  `ERROR: unable to allocate a new node for a list`
- **`push_front()`**: Adds an element to the front of the list exactly the same as `push_back()`.
- **`pop_back()`**: Removes an element from the back of the list, serving to reduce the size by one. Note that if the list is already empty, then the list remains unchanged.
- **`pop_front()`**: Removes an element from the front of the list exactly the same as `pop_front()`.
- **`erase()`**: Removes an element from the middle of a list. There is one parameter: a `list <T> :: iterator` pointing to the element to be removed. In the case of an iterator pointing to `end()`, then the method will simply return.
- **`find()`**: Takes a template element as a parameter and returns an iterator pointing to the corresponding element in the list. If the element does not exist, it returns `set::end()`.
- **`front()`**: Returns the element currently at the front of the list. There are two versions of this method: one that returns the element by-reference so the element can be changed through the `front()` method. If the list is currently empty, the following exception will be thrown:
  `ERROR: unable to access data from an empty list`
- **`back()`**: Returns the element currently at the back of the list exactly the same as `front()`.
- **`insert()`**: Inserts an element in the middle of a list. There are two parameters: the data element to be inserted, and a `list <T> :: iterator` pointing to the location in the list where the new element will be inserted before. The return value is an iterator to the newly inserted element. In the case of an allocation error, the following exception will be thrown:
  `ERROR: unable to allocate a new node for a list`
- **`begin()`**: Return an iterator to the first element in the list. It takes no parameters and returns a `list <T> :: iterator`.
- **`rbegin()`**: Return an iterator to the last element in the list. It takes no parameters and returns a `list <T> :: reverse_iterator`.
- **`end()`**: Return an iterator referring to the past-the-end element in the list. The past-the-end element is the theoretical element that would follow the last element in the container. It does not point to any element, so it must not be de-referenced.
- **`rend()`**: Return a reverse iterator referring to the past-the-front element in the list. The past-the-front element is the theoretical element that would preceed the first element in the container. It does not point to any element, so it must not be de-referenced.

Note that there is no square-bracket operator (`operator[]`) for the list. The only way to traverse the list is through an iterator.

Iterator

Additionally, create an iterator class that will traverse the list. Call this class `list <T> :: iterator`. Note that this iterator will work much like the iterator from Week 01 and Week 05. Your

iterator must be bi-directional. In other words, both of the following loops must work:

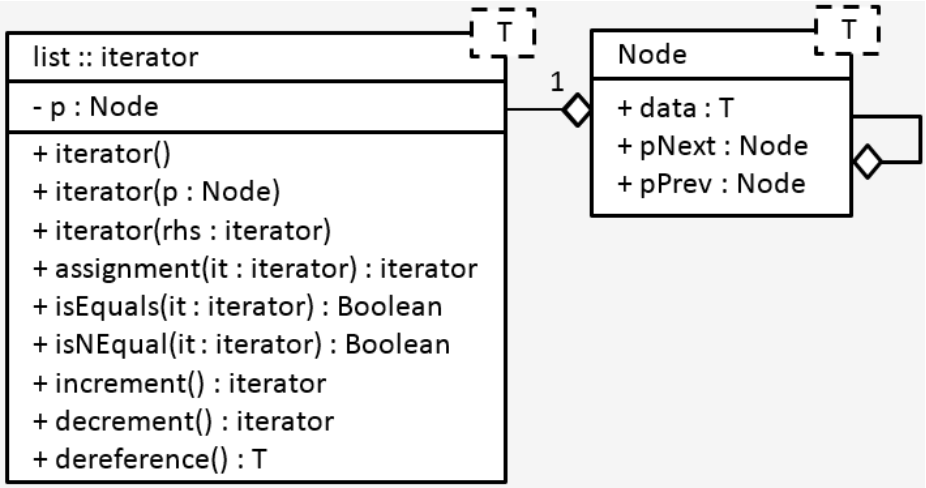
```
{
   // declare
   list <int> :: iterator it = l.begin();

   // forward with ++
   it++;
   ++it;

   // backwards with --
   it--;
   --it;
}
```

The STL uses both forward and reverse iterators. We will need to do the same:

```
for (list <int> :: iterator it = l.begin(); it != l.end(); ++it)
   cout << *it << endl;
for (list <int> :: reverse_iterator it = l.rbegin(); it != l.rend(); ++it)
   cout << *it << endl;
```

In the above example, observe that it is necessary to use the increment operator on the reverse iterator to go backwards.

The UML class diagram of `list <T> :: iterator` is:



Driver Program

A driver program is provided. This file (`/home/cs235/week07/assignment07.cpp`) will pound-include your header file (`list.h`) and expect a template class `List` to be defined therein. This program will exercise your class, filling the container with user input and displaying the results. As with previous assignments, a makefile will be provided (`/home/cs235/week07/makefile`) as well as a header file (`fibonacci.h`) and an implementation file (`fibonacci.cpp`). You will need to provide the list header file (`list.h`).

Fibonacci

The Fibonacci sequence is a sequence of numbers defined in such a way that the first element is 1, the second element is 1, and the third and subsequent elements are the sum of the previous two elements. Thus the first ten numbers in the Fibonacci sequence are

```
{ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 }
```

Of course this can be computed recursively, but that is an $O(2^n)$ algorithm:

```
int fibonacci(int num)
{
   if (num == 1 || num == 2)
      return 1;
   return fibonacci(num - 1) + fibonacci(num - 2);
}
```

Your Fibonacci algorithm must be performed in O(n) to get full credit.

There is one caviot to this problem: Your program must be able to output very large Fibonacci numbers:

```
Which Fibonacci number would you like to display? 100
        354,224,848,179,261,915,075
```

To see how this works, consider the following execution:

```
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? 100
        354,224,848,179,261,915,075
```

To accomplish this, you will need to create your own data-type that is able to handle numbers of arbitrary length. This data-type will need to use your `list` class where each node is a three-digit grouping of numbers. The above value will be seven nodes. In other words, integers can only represent numbers up to 4 billion (11 digits) whereas the above number has 21 digits. As you can imagine, your new data-type will need to be able to perform addition, copy/assignment, and display. Make this class support the following operators:

- `<<`: The insertion operator will be needed to display the number on the screen. Note that it will need to insert the thousands separator (,) every third digit
- `+=`: The add-onto operator will need to add two `WholeNumber`s and put the results in `this`. To accomplish this, you will need to add each bucket of digits (a `Node`) one at a time, making sure to handle the carry case. In other words, you will need to use a technique similar to what you used in elementary school.
- `=`: Copy one `WholeNumber` into a second. This should be trivial; use the assignment operator from the `list` class.
- Constructors: Create a default constructor, the copy constructor, and a non-default constructor taking an `unsigned int` as a parameter.

Test this before inserting it into your Fibonacci algorithm. This will vastly simplify the problem.

A few hints that may come in handy when implementing this part of the assignment:

- Implement the Fibonacci number first with integers to make sure it works. Only after you are sure you got the algorithm working should you attempt to handle large numbers.
- The `list <T> :: iterator` class will be quite different than anything we have done before. To solve this problem, compare the standard FOR loop for arrays with the standard FOR loop for linked-lists:

```
{
   for (int i = 0; i < num; i++)
```

```
      cout << array[i] << endl;
   for (Node * p = pHead; p; p = p->pNext)
      cout << p->data << endl;
}
```

- Do not forget to remove all the nodes in the linked-list when the object is destroyed.
- The `insert()` and `erase()` methods take iterators as parameters. These indicate where a node is to be added or removed from the linked-list. Unfortunately, these methods need to get a pointer to the `Node` in order to work. The `list <T> :: iterator` method does not provide a way to reach a `Node *`, only a `T`. To make this work, you need to make `list <T> :: insert()` and `list <T> :: erase()` friends to `list <T> :: iterator`.

As with the previous lessons, you must use your own `list` class to get full credit. If your class does not work, use the standard template library `std::list` from `#include <list>`. If you do this, you will loose points for the first half of the assignment, but not the second.

Common Mistakes

The most common mistakes students make with this assignment include the following:

- **Carry**: When performing addition with the `WholeNumber` class, you will need to handle the case when the value in a node is greater than 999. In that case, you will have to carry the number over to the next block of digits. This may involve creating a new node.
- **Delete from List**: It is tricky to delete a node from a linked list. You will need to be very careful of the order in which you reassign pointers or you might break your linked list chain.
- **Deleting the List**: When calling the `clear()` method or when the destructor is called, it will be necessary to delete all the nodes in the linked list. Make sure all the nodes are properly deleted; do not just set the pointer to `NULL` and forget about them!

One final note. Inserting the `Node` class as a nested class into `list` might be somewhat tricky. You will need to use the `typename` keyword in a few places to get the compiler to recognize the nested class. If this proves impossible to figure out, then implement the `Node` as a non-nested class.

Test Bed

The testBed for this assignment is:

```
testBed cs235/assign07 assignment07.tar
```

You can also run testBed on the executable:

```
testBed cs235/assign07 a.out
```

Of course, you will need to pass testBed to get full credit on the assignment.

Submitting

You will submit this assignment as a pair using the Linux submit command. Please:

1. Create a TAR file built from the makefile, which will contain at least five files:
   - `makefile`: Directly from `/home/cs235/week07/makefile` except with your edits on the comment block.
   - `list.h`: Your class definition for `list`.
   - `fibonacci.h`: Containing the prototype for `fibonacci()` and any other functions or classes you may need.

- `fibonacci.cpp`: Implementation for all the functions and classes necessary for the Fibonacci program.
- `assignment07.cpp`: Unmodified from `/home/cs235/week07/assignment07.cpp`.

2. Run the program by hand a few times through all four test cases as well as the Fibonacci program.
3. Verify your solution with testBed.
4. Submit your file using the `submit` command. The `submit` command will prompt you for your instructor, the class (`cs235`), and the assignment (`assign07`). You submit your file with:

```
submit assignment07.tar
```

Your program will be graded according to the following rubric:

	Exceptional 100%	Good 90%	Acceptable 70%	Developing 50%	Missing 0%
List Interface 20%	The interfaces are perfectly specified with respect to const, pass-by-reference, etc.	`assignment07.cpp` compiles without modification	All of the methods in `list` and `list <T> :: iterator` 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
List Implementation 20%	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
Whole Numbers 30%	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
Fibonacci 10%	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
Code Quality 10%	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
Style 10%	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

Please make sure to fill out the program header in the makefile with the following information: the name of both programmers, the amount of coding time required by each to complete the

assignment, the amount of discussion time (the Pair Programming part), and what was the most difficult part. Failure to do this will result in a loss of 10% on the assignment.

In addition to the above criteria, the following extra credit opportunities exist:

1. **5%**: Implement a `const_iterator` and `const_reverse_iterator`.
2. **5%**: Extend the `WholeNumber` class to include subtraction.
3. **10%**: Extend the `WholeNumber` class to include the extraction operator.
4. **10%**: Extend the `WholeNumber` class to include multiplication.