Destructor Prototype: ~Table();

Function:

```
template <class T> void Table<T>::~Table() {
  for (unsigned int j = 0; j < rows; j++) {
    delete [] values[j];
  }
  delete [] values;
}
```

Templated Table Example:

```
template <class T> class Node {
public:
		//Default construtor.
		Node(){
				//Set defaults for values.
				cSize = 0;
				next_ = NULL;
				prev_ = NULL;
				value_ = new T[NUM_ELEMENTS_PER_NODE];
		}
		Node(const T& v){
				next_ = NULL;
				prev_ = NULL;
				value_ = new T[NUM_ELEMENTS_PER_NODE];
				value_[0] = v;
				cSize = 1;
		}
```

Stream Manipulators:

```
#include <iostream>, std::cout, std::cin
std::cout << std::endl; // ends line in output stream, clears buffer
std::cin >> var_name >> var_name2;
std::setprecision(); //requires std::fixed
```

Copy Constructor Prototype:

```
Table(const Table& t) { copy(t); }
```

Assignment Operator Prototype:

```
const Table& operator=(const Table& t);
```

Functions:

```
//Assgn 1 Tble 2 another, avoid self-assgnment
template <class T> const Table<T>&
Table<T>::operator=(const Table<T>& v) {
  if (this != &v) {
    destroy();
    this->copy(v); //Copy is below
  }
  return *this;
}
//Create the Tble as a copy of the given Tble
template <class T> void Table<T>::copy(const Table<T>& v)
{
  this->create(v.rows,v.cols);
  for (unsigned int i = 0; i < rows; i++) {
    for (unsigned int j = 0; j < cols; j++) {
      values[i][j] = v.values[i][j];
    }
  }
}
```

Const(antly screwing up consts):

-- Const objects can only be used by const member functions

-- In classes, if const at end of member function prototype then it does not change any member variables.

Standard Library Types (and useful tricks):

Char: Designated by single quotes, just a character.

Int: Woah… a whole number. Nothing fancy, gets the job done.

Float/Double: DON'T FORGET ABOUT THESE. Sorry for no tricks.

Order Notation:

-- O(1), a.k.a. CONSTANT: The number of operations is independent of the size of the problem. e.g., compute quadratic root.
-- O(log n), a.k.a. LOGARITHMIC. e.g., dictionary lookup, binary search.
-- O(n), a.k.a. LINEAR. e.g., sum up a list.
-- O(n log n), e.g., sorting.
-- O(n^(1/2)), O(n^3), O(n^k), a.k.a. POLYNOMIAL, find the closest pair
-- O(2^n), O(kn), a.k.a. EXPONENTIAL. e.g., Fibonacci, playing chess.
-- O(N * M), nested for loops.

Iterators (Abed)/Reverse Iterators (Evil Abed):

-- use dereference operator to access value at iterator (*)
-- use select/dereference operator to access member functions (itr->member()).
-- reverse_iterator increments backwards, find beginning reverse itr with .rbegin() and the .rend().
--*itr for value
-- itr->func() is the same as (*itr).func()

(How to abuse the) Sort (function and get away with it):

```
#include <algorithm>
//function prototype for sorting & sort call example
bool by_total_snowfall(const Snow &a, const Snow &b);
sort(container.begin(), container.end(),by_total_snowfall);
```

STD::FIND:

```
#include <algorithm>
std::find(container.begin(), container.end(), value);
```

Standard Library Containers:

Arrays: Can be dynamically created, fixed size, has [], created by type[size], int t[] = {4,5,3,2,2}, has size, iterator stuff, etc.

std::string: Container of chars, has iterator stuff, size(), [], can append with +=, push_back/pop_back, insert, erase.

std::vector: Has [], push/pop_back, insert, eras, and iterator stuff. Can access iterator with v.begin() + int.

std::list: Has iterator stuff, push/pop _ back/front, .front() and .back() for element access, no []! Not connected

Erase & Insert:

```
var.erase(iterator position);
//erases the object at position, returns next
var.insert(iterator position, val);
//inserts val in container before position
container<type>::iterator for itr
```

Recursion Example:

```
int intpow(int n, int p) {
   if (p == 0) {
      return 1;
   } else {
      return n * intpow(n, p-1);
   }
}
void countdown(int n) {
   std::cout << n << std::endl;
   if (n == 0) return;
   else countdown(n-1);
}
```

Operators:

`+,-,*, /, %, >, <, !=, ==, +=, -=,*=, /=, %=`

`Also! Don't forget you can ++i and --i.`

Assignment Operator Special: (:)

```
TrainCar(char t, int w) : type(t), weight(w), prev(NULL){
    //other function stuff can go here
}
```

Recursive Print Data:

```
void PrintData(Node *head) {
    if (head == NULL) return; //(!head) works
    std::cout << head->value << " ";
    PrintData(head->next);
}
```

Vector Push Front:

```
template <class T>
void Vec<T>::push_front(const T& val) {
  // if it's the first element, use push_back
  if (m_alloc == 0) { push_back(val); return; }
  assert (m_alloc > 0);
  if (m_first == 0) {
    // Calculate the new allocation.
    m_alloc *= 2;
    assert (m_alloc > 1);
    // Allocate the new array
    T* new_data = new T[ m_alloc ];
    // put the existing data in the array
    m_first = m_alloc / 2;
    // copy the data
    for (unsigned int i=0; i<m_size; ++i) {
       new_data[m_first+i] = m_data[i]; }
    // delete the old array and reset
    delete [] m_data;
    m_data = new_data;
  }
  // move the first index back one spot
  m_first--;
  //Add the value at the end and increment
  m_data[m_first] = val;
  ++m_size;
}
```

Order Notation pt 2:

```
int foo(int n) {
   if (n == 1 || n == 0) return 1;
       return foo(n-1) + foo(n-2);
}
ans for above: O(2^n)
-----------------------------------------------
for (int i = 0; i < n; i++) {
   my_vector.erase(my_vector.begin());
}
ans for above: O(n^2) (erase loops through too)
```

Lab:
2-3:50 Lally 104, Mauricio, Alec, Matt

```
            list_iterator<T>& operator++() { // pre-
increment, e.g., ++iter
            if (index == ptr_->cSize -1){//Check if
index is at the end of
                  // the node and if so jump to the
next one. Otherwise iterate it.
                  index = 0;
                  ptr_ = ptr_->next_;
            }
            else{
                  index++;
            }
            return *this;
      }list_iterator() : ptr_(NULL) {}
      list_iterator(Node<T>* p) : ptr_(p) {}
      list_iterator(const list_iterator<T>& old, int
indeI) : ptr_(old.ptr_), index (indeI) {}
      list_iterator(const list_iterator<T>& old) :
ptr_(old.ptr_), index (old.index) {}
      list_iterator<T>& operator=(const list_iterator<T>&
old) {
            ptr_ = old.ptr_;  return *this; }
      ~list_iterator() {}

      T& operator*()  {return ptr_->value_[this->index]; }
```

Template Class Example:

```
#ifndef Vec_h_
#define Vec_h_
template <class T> class Vec {
public:
  // TYPEDEFS (two redacted)
  typedef unsigned int size_type;
  // CONSTRUCTORS, ASSIGNMNENT OPERATOR, & DESTRUCTOR
  Vec() { this->create(); }
  Vec(size_type n, const T& t = T()) { this->create(n, t); }
  Vec(const Vec& v) { copy(v); }
  Vec& operator=(const Vec& v);
  ~Vec() { delete [] m_data; }
  // MEMBER FUNCTIONS AND OTHER OPERATORS
  T& operator[] (size_type i) { return m_data[i]; }
  const T& operator[] (size_type i) const { return m_data[i]; }
  void push_back(const T& t);
  iterator erase(iterator p);
  void resize(size_type n, const T& fill_in_value = T());
  void clear() { delete [] m_data;  create(); }
  bool empty() const { return m_size == 0; }
  size_type size() const { return m_size; }
  // ITERATOR OPERATIONS
  iterator begin() { return m_data; }
  const_iterator begin() const { return m_data; }
  iterator end() { return m_data + m_size; }
  const_iterator end() const { return m_data + m_size; }
private:
  // PRIVATE MEMBER FUNCTIONS
  void create();
  void create(size_type n, const T& val);
  void copy(const Vec<T>& v);
  // REPRESENTATION
  T* m_data;         // Pointer to first location in the
allocated array
  size_type m_size;  // Number of elements stored in the vector
  size_type m_alloc; // Number of array locations allocated,
m_size <= m_alloc
};
// Create an empty vector (null pointers everywhere).
template <class T>  void Vec<T>::create() {
  m_data = NULL;
  m_size = m_alloc = 0;  // No memory allocated yet
}
// Create a vector with size n, each location having the given
value
template <class T> void Vec<T>::create(size_type n, const T& val)
{
  m_data = new T[n];
  m_size = m_alloc = n;
  for (T* p = m_data; p != m_data + m_size; ++p)
    *p = val;
}
// Shift each entry of the array after the iterator. Return the
iterator,
// which will have the same value, but point to a different
element.
template <class T> typename Vec<T>::iterator
Vec<T>::erase(iterator p) {
  // remember iterator and T* are equivalent
  for (iterator q = p; q < m_data+m_size-1; ++q)
    *q = *(q+1);
  m_size --;
  return p;
}
// If n is less than or equal to the current size, just change
the size.  If n is
// greater than the current size, the new slots must be filled in
with the given value.
template <class T> void Vec<T>::resize(size_type n, const T&
fill_in_value) {
  if (n <= m_size)
    m_size = n;
  else {
    // If necessary, allocate new space and copy the old values
    if (n > m_alloc) {
      m_alloc = n;
      T* new_data = new T[m_alloc];
      for (size_type i=0; i<m_size; ++i)
        new_data[i] = m_data[i];
      delete [] m_data;
      m_data = new_data;
    }
    // Now fill in the remaining values and assign the final
size.
    for (size_type i = m_size; i<n; ++i)
      m_data[i] = fill_in_value;
    m_size = n;
  }
}
#endif
```